

Enhancing Host-Based Security: Leveraging Hardware and Compiler Techniques for Robust Protection

AFSecurity seminar

Guillaume Hiet

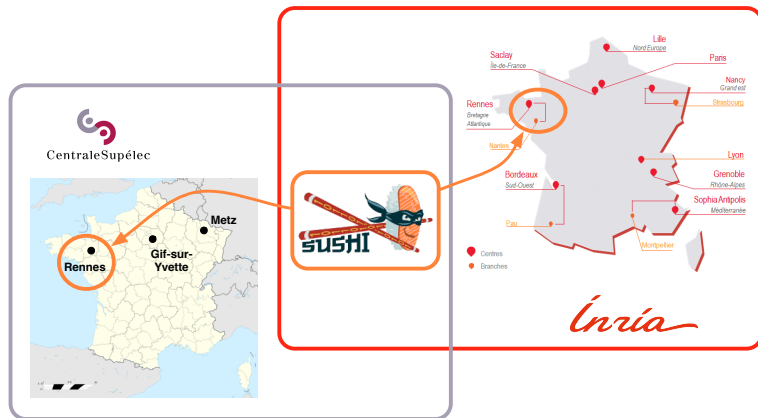
<https://guillaume.hiet.fr/>

CentraleSupélec, SUSHI Inria Project-Team



November 28th, 2024

SUSHI: a joint team between CentraleSupélec and Inria



<https://team.inria.fr/sushi/>



Context

Cyber-security is a major concern

- Many vulnerable systems are targeted by sophisticated attacks
- Linked to underground economy and military/intelligence activities

Security Properties

- **Confidentiality**: information is not disclosed to unauthorized entities
- **Integrity**: data cannot be modified in an unauthorized manner
- **Availability**: attackers cannot prevent or disrupt legitimate access to data

Attacks

- Attempts to exploit some vulnerability to violate security properties.

Context

Cyber-security is a major concern

- Many vulnerable systems are targeted by sophisticated attacks
- Linked to underground economy and military/intelligence activities

Security Properties

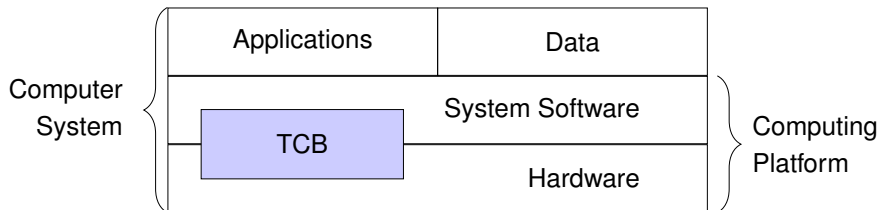
How to protect such systems from **intrusions**, i.e. violations of the security properties resulting from successful attacks?

- **Availability**: attackers cannot prevent or disrupt legitimate access to data

Attacks

- Attempts to exploit some vulnerability to violate security properties.

Our main topic: security of computing platforms



Computing Platform = Hardware + System Software

- System Software = OS, hypervisor, firmware
- The Computing Platform implements the **Trusted Computing Base**
- Objectives : optimize performance, offer new services while limiting energy consumption
 - Increasingly **complex** interactions between software and heterogeneous computation units (CPU, GPU, NPU, FPGA, etc.)

Security at the software/hardware interface

Combining hardware and software for the security of Computing Platforms

- Some attacks (e.g., Spectre) exploit both hardware and software weaknesses
- Opportunities to codesign hardware/software security mechanisms
 - Enclaves, security coprocessors, security extensions of the ISA, etc.

Research question: how can we assess and improve the security of computing platforms?

- What **vulnerabilities** result from software/hardware interactions and how to **avoid** them?
- How to **detect** and **react** to intrusion at the host-level?
- How to **prove** hardware/software security mechanisms?

Members of the research group

Faculty members and permanent researcher

- Guillaume Hiet, Prof. CentraleSupélec, team leader
- Alessandro Palumbo, Assoc. Prof. CentraleSupélec
- Thomas Rokicki, Assoc. Prof. CentraleSupélec
- Louis Rilling, Research Engineer DGA-MI (40% at Inria, Creach labs)
- Rubén Salvador, Assoc. Prof. CentraleSupélec
- Frédéric Tronel, Assoc. Prof. CentraleSupélec
- Yaëlle Vinçont, Assist. Prof. (AGPR) ENS Rennes
- Pierre Wilke, Assoc. Prof. CentraleSupélec

PhD students and postdocs

- Postdocs: Quentin Ducasse & Lorenzo Casalino
- PhD students: Lionel Hemmerlé, Matthieu Batty, Eliott Quéré, Erwan Fasquel

Members of the research group

Faculty members and permanent researcher

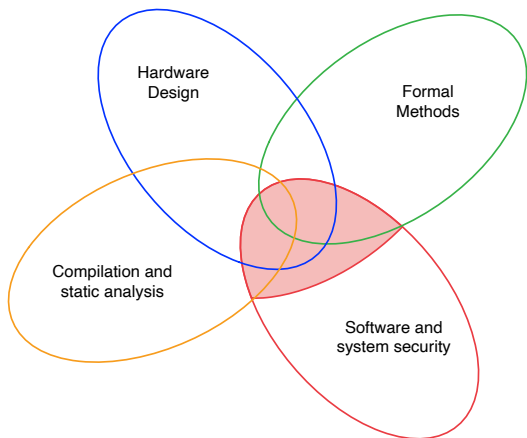
Guillaume Llet, Prof. Centrale, Supervisor, team leader

We are hiring!

- M2 internship (and future PhD)
- Postdocs in formal methods for hardware/software security
- Tenured associate and full professor positions (should be officially announced soon, deadline: end of March 2025)
- Tenured CNRS and Inria full researcher positions (competitive exam, deadline: early February 2025)
- Not all positions have been posted yet; feel free to contact me if you're interested in the team's topics.

- PhD students: Lionel Hemmerlé, Matthieu Batty, Elliott Quéré, Erwan Fasquel

My Research Areas



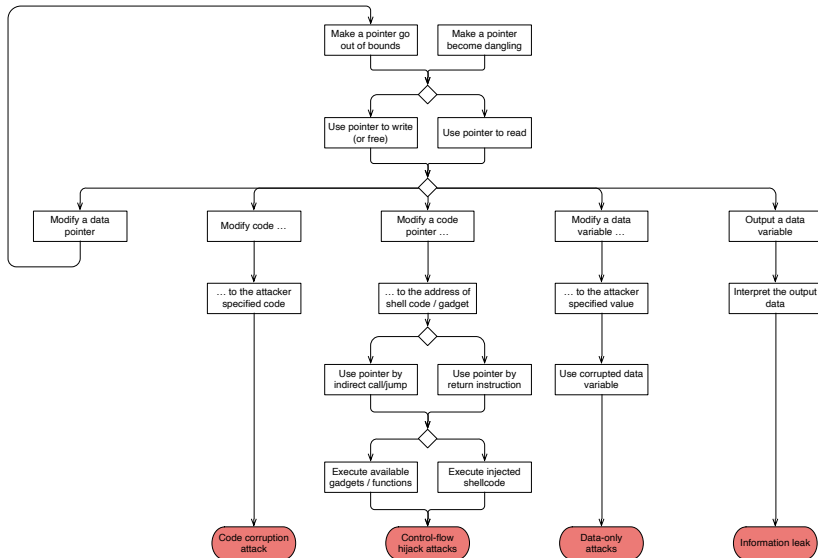
Originality

- We **combine** hardware and compilation to contribute to **(low-level) software security**.
- We use formal methods to reason on the security of **interactions between hardware and software**.

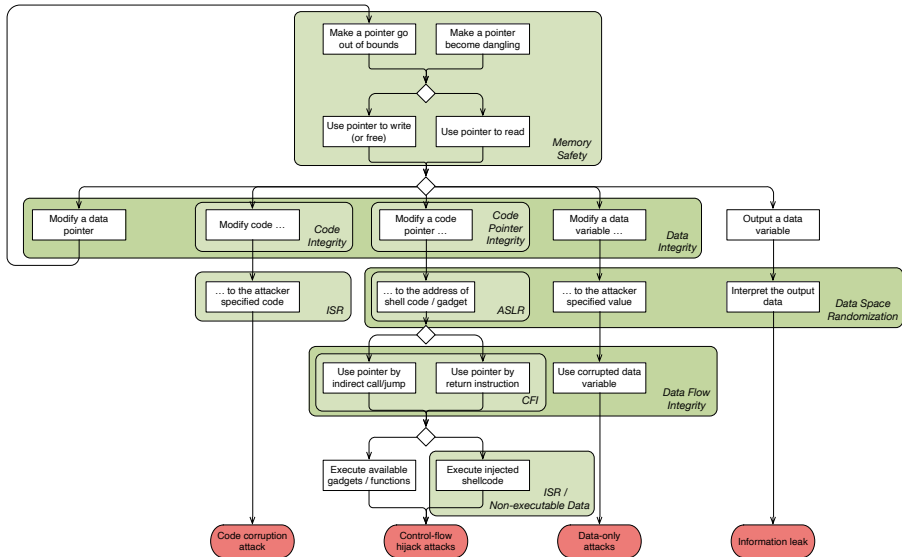
Threat model

- We only consider **software attacks**
- We do not consider physical attacks

Memory corruption vulnerabilities

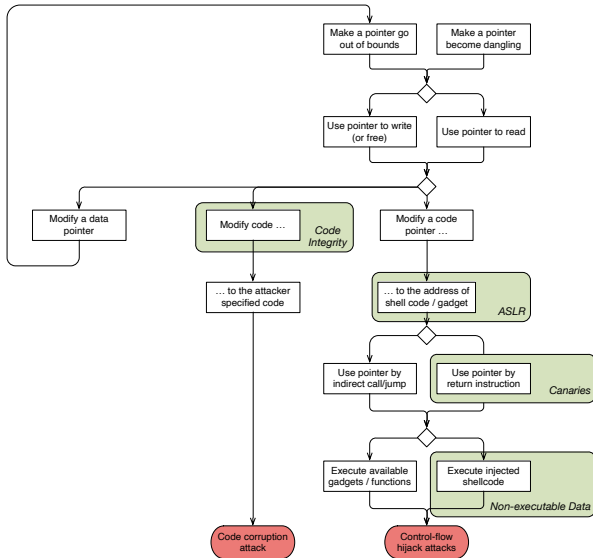


Memory corruption vulnerabilities



SZEKERES et al. "SoK : Eternal War in Memory", IEEE S&P 2013

Memory corruption vulnerabilities



SZEKERES et al. "SoK : Eternal War in Memory", IEEE S&P 2013

Limitations of currently deployed protections

$W \oplus X$

- JIT code
- Does not protect against code-reuse attacks (e.g. ROP or return-into-libc)

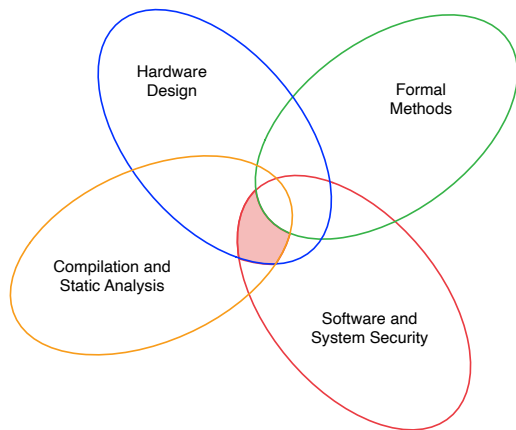
Address Space Randomization

- Derandomization is possible if entropy is too low
- Addresses can be leaked

Canaries

- Only protect the return address
- Only efficient for stack smashing attacks
- Can be bypassed by direct overwrite (e.g. using indexing error)

Combining Hardware and Compilation to Detect Intrusions against the BIOS



Ph.D. thesis of Ronny Chevalier

Joint work with Ronny Chevalier, David Plaquin, and Chris Dalton from HP Labs



Host-based Intrusion Detection Challenges

How to protect the monitor?

- HIDS are co-localized with the monitored target
- We have to **isolate** the monitor from the potentially infected host
- Isolation creates a **semantic gap**: how to retrieve useful information from the monitored host?

How to detect intrusions?

- Misuse-based approaches rely on signatures of attacks
 - Signatures have to be updated frequently
- **Anomaly-based** approaches rely on a model of legitimate behaviour
 - The model must be complete to prevent false positives

Positioning of our Approach

Why using hardware?

- Hardware features cannot be modified by software attacks
- The security mechanisms expose a reduced and controlled interface

Security coprocessors

- Enclaves (e.g., SGX, Intel VT): not always available + limited isolation
- Coprocessors → no modification of the main CPU
- Snapshot-based approaches [Zhang 2002; Petroni 2004; Petroni 2006; Wang 2010] are vulnerable to **transient attacks**
- Most event-based approaches [Moon 2012; Lee 2013; Zhou 2021] focus only on code integrity

First approach to detect intrusions against the BIOS at runtime.

Introducing the System Management Mode (SMM)

SMM is the **highest privileged** execution mode for x86 processors.

How to enter the SMM?

- Trigger a System Management Interrupt (SMI)
- SMIs code & data are stored in a **protected memory region**: System Management RAM (SMRAM)

SMM and UEFI BIOS

- The BIOS is in charge of the initialisation of the platform
- SMM isolates critical BIOS **runtime services** while the OS is running:
 - BIOS update, power management, UEFI variables handling, etc.

Security of SMI Handlers

Why compromising the BIOS?

- Malware can be hard to detect (**stealth**)
- Malware can be **persistent** (survives even if the HDD/SSD is changed)

Why targeting SMI handlers?

- They exchange data with the OS
- Only CPU in SMM can write to the flash containing the BIOS
- Arbitrary code execution in SMM gives **full control of the platform**

Security risks

- Vulnerabilities in SMM code [Bazhaniuk 2015; Bulygin 2017; Pujos 2016]
- **Boot-time** security (e.g. Intel Boot Guard, signed update) is **insufficient**
 - It does not protect SMI handlers between two (infrequent) consecutive reboots

Security of SMI Handlers

Why compromising the BIOS?

- Malware can be hard to detect (**stealth**)
- Malware can be **persistent** (survives even if the HDD/SSD is changed)

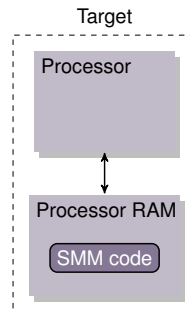
Why targeting SMI handlers?

We have to monitor the **behavior** of SMI handlers at **runtime** to detect intrusions targeting the SMM

Security risks

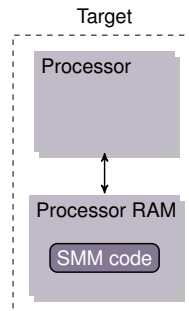
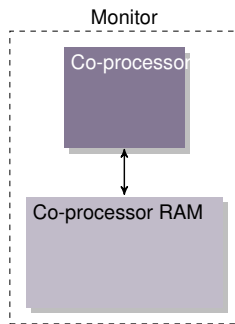
- Vulnerabilities in SMM code [Bazhaniuk 2015; Bulygin 2017; Pujos 2016]
- **Boot-time** security (e.g. Intel Boot Guard, signed update) is **insufficient**
 - It does not protect SMI handlers between two (infrequent) consecutive reboots

Approach Overview



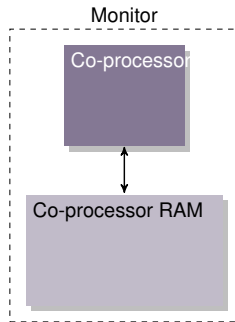
Approach Overview

How to ensure the integrity of the monitor?

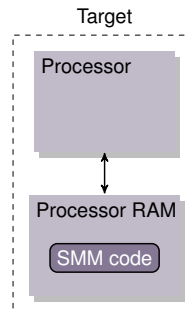


Approach Overview

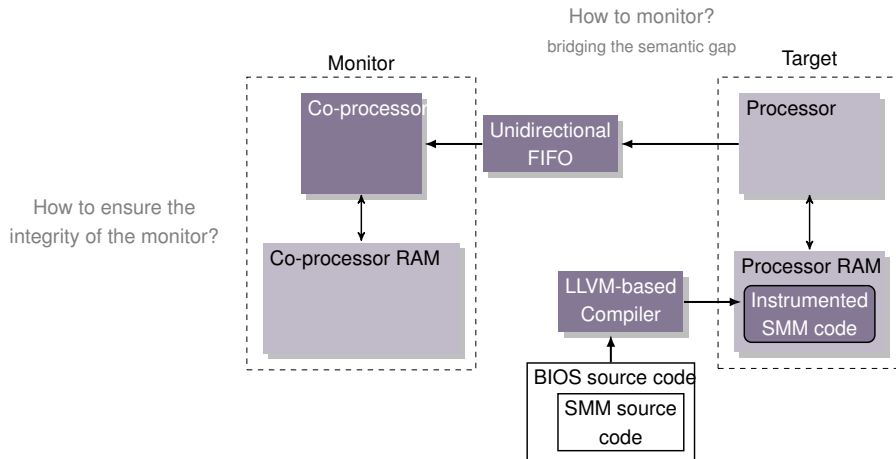
How to ensure the integrity of the monitor?



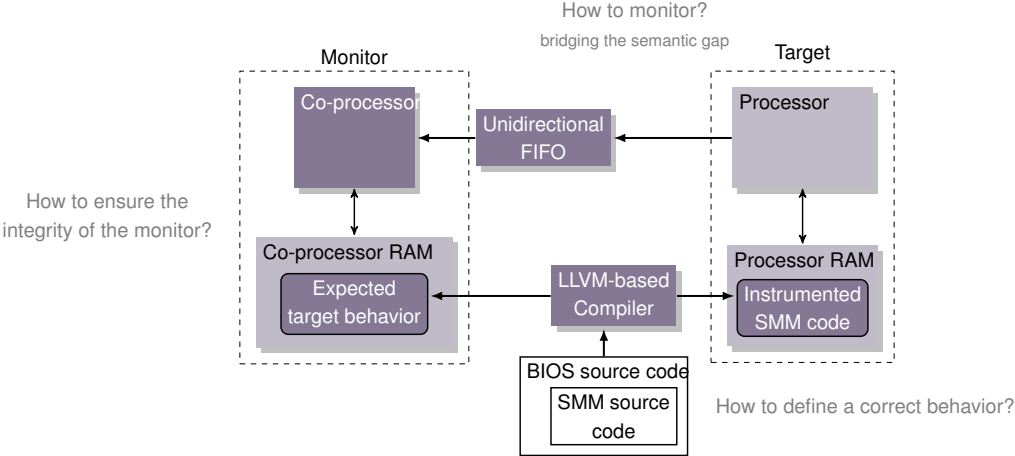
Semantic gap?



Approach Overview



Approach Overview



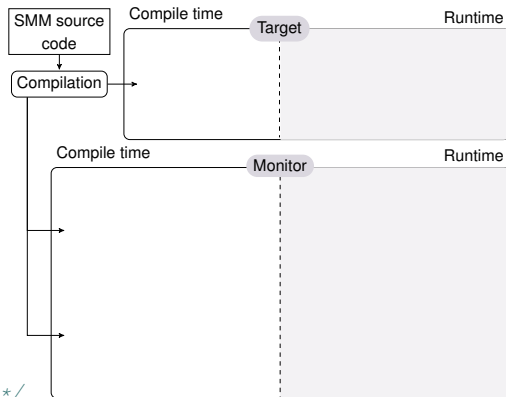
Type-based Control Flow Integrity (CFI)

We focus on indirect branches

We assume code is read-only (using pagination)

Type-based verification of indirect calls

```
typedef struct SomeStruct {  
    [...]   
    char (*foo)(int);  
} SomeStruct;  
  
int bar(SomeStruct *s) {  
    char c;  
    [...]  
  
    c = s->foo(31); /* Call Site ID = 1561 */  
    [...]  
}
```



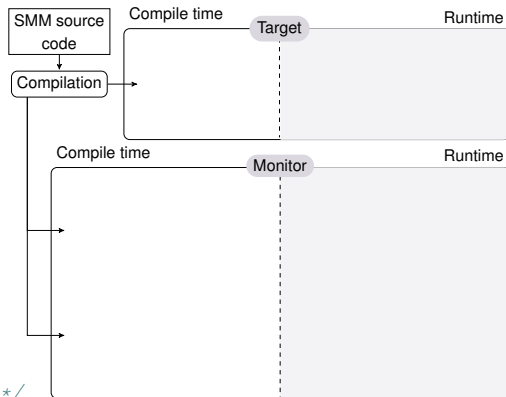
Type-based Control Flow Integrity (CFI)

We focus on indirect branches

We assume code is read-only (using pagination)

Type-based verification of indirect calls

```
typedef struct SomeStruct {  
    [...]   
    char (*foo)(int);  
} SomeStruct;  
  
int bar(SomeStruct *s) {  
    char c;  
    [...]  
  
    c = s->foo(31); /* Call Site ID = 1561 */  
    [...]  
}
```



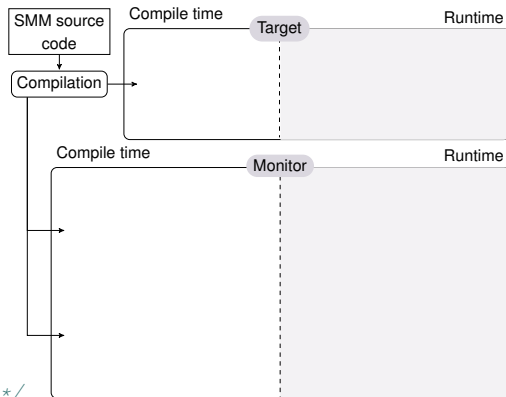
Type-based Control Flow Integrity (CFI)

We focus on indirect branches

We assume code is read-only (using pagination)

Type-based verification of indirect calls

```
typedef struct SomeStruct {  
    [...]   
    char (*foo)(int);  
} SomeStruct;  
  
int bar(SomeStruct *s) {  
    char c;  
    [...]  
  
    c = s->foo(31); /* Call Site ID = 1561 */  
    [...]  
}
```



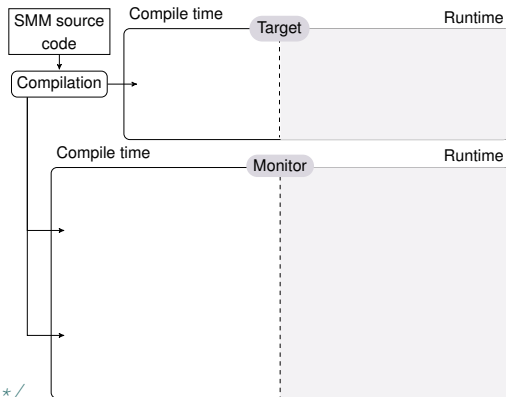
Type-based Control Flow Integrity (CFI)

We focus on indirect branches

We assume code is read-only (using pagination)

Type-based verification of indirect calls

```
typedef struct SomeStruct {  
    [...]   
    char (*foo)(int);  
} SomeStruct;  
  
int bar(SomeStruct *s) {  
    char c;  
    [...]  
  
    c = s->foo(31); /* Call Site ID = 1561 */  
    [...]  
}
```



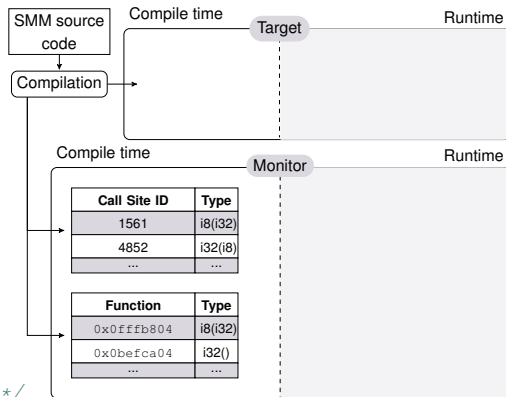
Type-based Control Flow Integrity (CFI)

We focus on indirect branches

We assume code is read-only (using pagination)

Type-based verification of indirect calls

```
typedef struct SomeStruct {  
    [...]   
    char (*foo)(int);  
} SomeStruct;  
  
int bar(SomeStruct *s) {  
    char c;  
    [...]  
  
    c = s->foo(31); /* Call Site ID = 1561 */  
    [...]  
}
```



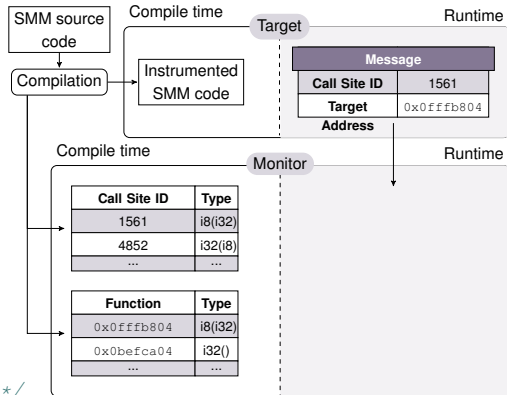
Type-based Control Flow Integrity (CFI)

We focus on indirect branches

We assume code is read-only (using pagination)

Type-based verification of indirect calls

```
typedef struct SomeStruct {  
    [...]   
    char (*foo)(int);  
} SomeStruct;  
  
int bar(SomeStruct *s) {  
    char c;  
    [...]   
  
    c = s->foo(31); /* Call Site ID = 1561 */  
    [...]   
}
```



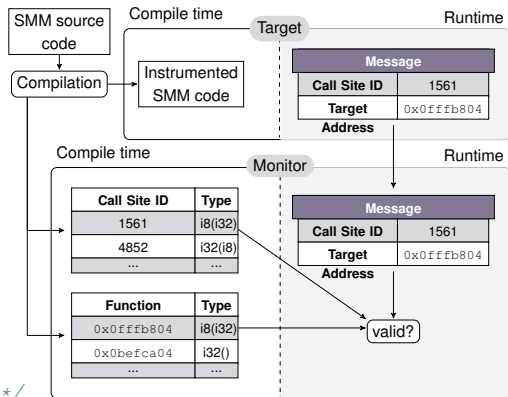
Type-based Control Flow Integrity (CFI)

We focus on indirect branches

We assume code is read-only (using pagination)

Type-based verification of indirect calls

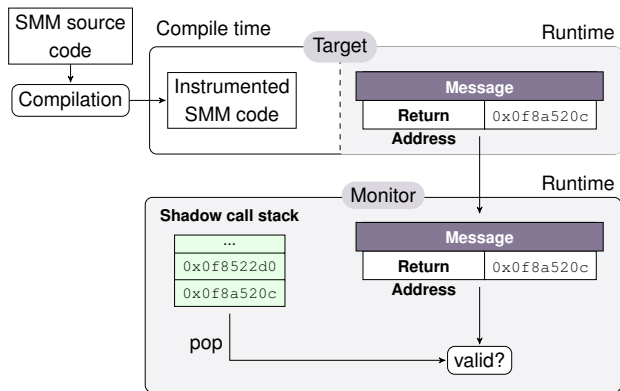
```
typedef struct SomeStruct {  
    [...]   
    char (*foo)(int);  
} SomeStruct;  
  
int bar(SomeStruct *s) {  
    char c;  
    [...]   
  
    c = s->foo(31); /* Call Site ID = 1561 */  
    [...]   
}
```



How to define a correct behaviour?

Shadow call stack

Ensures integrity of the return address on the stack



Experimental Setup

Our prototype is implemented in a simulated and emulated environment

SMM code implementations used

- EDK2: the foundation of many BIOSes (Apple, HP, Intel,...)
→ UEFI Variables SMI handlers
- coreboot: perform hardware initialization (used on some Chromebooks)
→ Hardware-specific SMI handlers

We want to emulate SMM environment and features

QEMU emulator for security evaluation

We want to simulate accurately the performance impact

gem5 simulator for performance evaluation

Security Evaluation

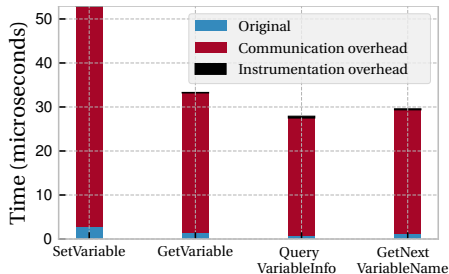
We simulated attacks that exploited vulnerabilities similar to those found in real-world BIOSes

Vulnerability	Attack Target	Security Advisories	Detected
Buffer overflow	Return address	CVE-2013-3582	Yes
Arbitrary write	Function pointer	CVE-2016-8103	Yes
Arbitrary write	SMBASE	LEN-4710	Yes
Insecure call	Function pointer	LEN-8324	Yes

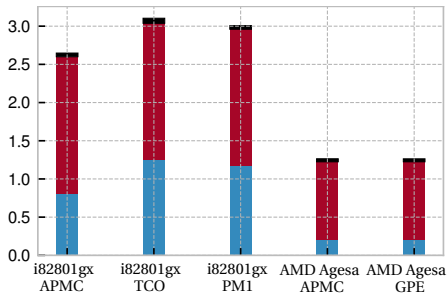
Runtime Overhead for SMI Handlers

- Under the 150 microseconds limit defined by Intel
- Most of the communication overhead is due to the shadow call stack

EDK2



coreboot



Limitations

Detection model

- Type collisions are not detected
- Non-control data attacks are not detected

Execution overhead

- Mainly due to the shadow call stack (protection of return address)
- We could rely on Intel CET to protect return addresses
- We could also rely on static analysis to prevent instrumentation if there is no risk

Hardware design

- Approach only evaluated in simulation
- Implementing a dedicated FIFO using low latency interconnect is costly and challenging

Conclusion on Hardware-assisted Intrusion Detection

- UEFI runtime services executed in **SMM** can be vulnerable to memory errors
- Attackers exploiting such vulnerabilities can gain full control of the platform

Conclusion on Hardware-assisted Intrusion Detection

- UEFI runtime services executed in **SMM** can be vulnerable to memory errors
- Attackers exploiting such vulnerabilities can gain full control of the platform
- We can detect such attacks using a behavior and **event-based** intrusion detection system implemented on an **isolated** co-processor
- We can solve the **semantic gap** by relying on **static analysis** and **instrumentation** of the monitored target
- Overhead is acceptable ($< 150 \mu\text{s}$ Intel threshold), avoiding any noticeable impact on the user experience.

Conclusion on Hardware-assisted Intrusion Detection

- UEFI runtime services executed in **SMM** can be vulnerable to memory errors
- Attackers exploiting such vulnerabilities can gain full control of the platform
- We can detect such attacks using a behavior and **event-based** intrusion detection system implemented on an **isolated** co-processor
- We can solve the **semantic gap** by relying on **static analysis** and **instrumentation** of the monitored target
- Overhead is acceptable ($< 150 \mu\text{s}$ Intel threshold), avoiding any noticeable impact on the user experience.

More details in [ACSAC 2017, US11556645B2, US11308202B2]

Perspectives on Hardware-assisted Intrusion Detection

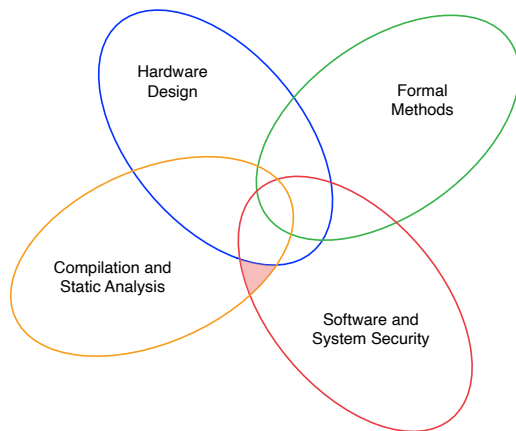
Exploring different detection approaches

- Exploring other detection approaches to cover a larger spectrum of attacks
- Attackers should pay more attention to non-control-data attacks

Monitoring heterogeneous systems

- Applications can offload part of their computations on hardware accelerators (e.g, FPGA)
- ANR TrustGW: monitoring of hybrid applications developed using High-Level Synthesis

Optimizing Data-Flow Integrity for Real-Time Systems



Ph.D. thesis of Nicolas Bellec

Joint work with Nicolas Bellec, Frédéric Tronel, Simon Rokicki and Isabelle Puaut

Real-time systems

System correctness depends on its response time

Real-time systems

System correctness depends on its response time



Real-time systems

System correctness depends on its response time



Real-time systems

System correctness depends on its response time



2-steps timing verification:

Real-time systems

System correctness depends on its response time



2-steps timing verification:

- Estimate the **Worst-Case Execution Time** (WCET) for each task

Real-time systems

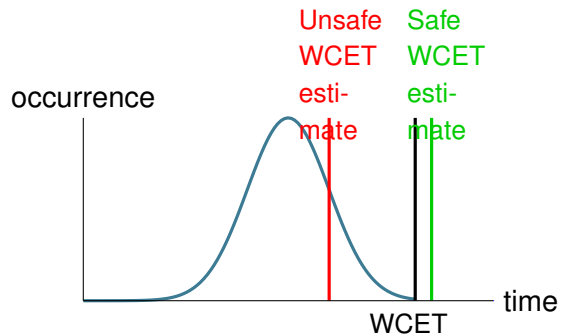
System correctness depends on its response time



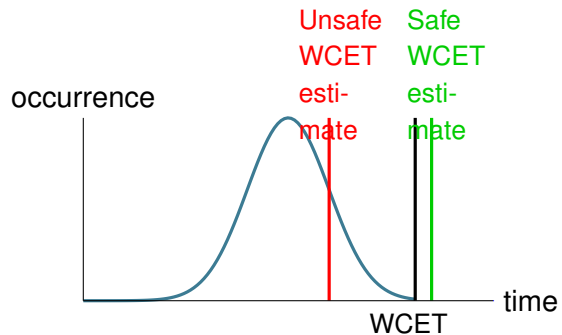
2-steps timing verification:

- Estimate the **Worst-Case Execution Time** (WCET) for each task
- Perform a **Schedulability Analysis**

WCET

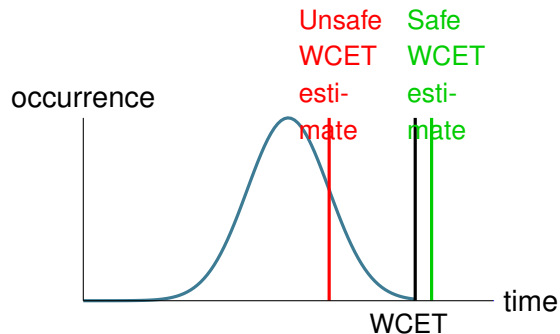


WCET



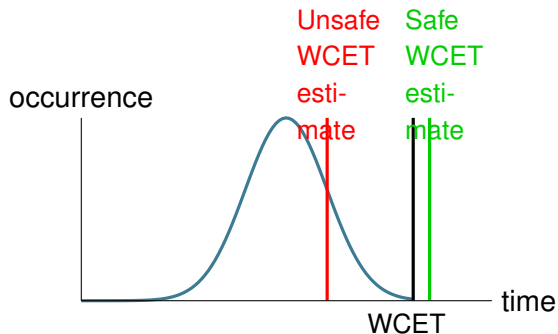
- Isolated tasks

WCET



- Isolated tasks
- Estimated on the binary with hardware architecture information

WCET



- Isolated tasks
- Estimated on the binary with hardware architecture information
- May require specific information (e.g. loopbounds)

Security for Real-Time Systems

Security for Real-Time Systems

- Increased complexity and wireless communications

Security for Real-Time Systems

- **Increased complexity and wireless communications**

Bluetooth, Wifi, IoT, ...¹

¹ *Remote exploitation of an Unaltered Passenger Vehicle*, Valasek et Miller, BlackHat'15

Security for Real-Time Systems

- **Increased complexity and wireless communications**

Bluetooth, Wifi, IoT, ...¹

- **Attackers break hypotheses**

¹ *Remote exploitation of an Unaltered Passenger Vehicle*, Valasek et Miller, BlackHat'15

Security for Real-Time Systems

- **Increased complexity and wireless communications**

Bluetooth, Wifi, IoT, ...¹

- **Attackers break hypotheses**

Deadline miss $\Rightarrow$



¹ Remote exploitation of an Unaltered Passenger Vehicle, Valasek et Miller, BlackHat'15

Security for Real-Time Systems

- Increased complexity and wireless communications

Bluetooth, Wifi, IoT, ...¹

- Attackers break hypotheses

Deadline miss $\Rightarrow$



WCET estimated **statically** to have an upper bound

¹ Remote exploitation of an Unaltered Passenger Vehicle, Valasek et Miller, BlackHat'15

Security for Real-Time Systems

- Increased complexity and wireless communications

Bluetooth, Wifi, IoT, ...¹

- Attackers break hypotheses

Deadline miss $\Rightarrow$



WCET estimated **statically** to have an upper bound

Can trade some overhead for more safety

¹ Remote exploitation of an Unaltered Passenger Vehicle, Valasek et Miller, BlackHat'15

Real-Time Systems - Characteristics

Real-Time Systems - Characteristics

- **Unsafe low-level languages (C/C++)**

Real-Time Systems - Characteristics

- **Unsafe low-level languages (C/C++)**

Prone to Memory Corruption Attacks

Real-Time Systems - Characteristics

- **Unsafe low-level languages (C/C++)**

Prone to Memory Corruption Attacks

- **Hard to update**

Real-Time Systems - Characteristics

- **Unsafe low-level languages (C/C++)**

Prone to Memory Corruption Attacks

- **Hard to update**

Protection against unknown attacks

Real-Time Systems - Characteristics

- **Unsafe low-level languages (C/C++)**

Prone to Memory Corruption Attacks

- **Hard to update**

Protection against unknown attacks

- **Predictable**

Real-Time Systems - Characteristics

- **Unsafe low-level languages (C/C++)**

Prone to Memory Corruption Attacks

- **Hard to update**

Protection against unknown attacks

- **Predictable**

Protection overhead **on the WCET** must be predictable statically

Real-Time Systems - Characteristics

- **Unsafe low-level languages (C/C++)**

Prone to Memory Corruption Attacks

- **Hard to update**

Protection against unknown attacks

- **Predictable**

Protection overhead **on the WCET** must be predictable statically

Wanted

Robust and Predictable protection against Memory Corruption Attacks
for Real-Time Systems

Real-Time Systems - Characteristics

- **Unsafe low-level languages (C/C++)**

Prone to Memory Corruption Attacks

- **Hard to update**

Protection against unknown attacks

- **Predictable**

Protection overhead **on the WCET** must be predictable statically

Wanted

Robust and Predictable protection against Memory Corruption Attacks
for Real-Time Systems

Real-Time Systems - Characteristics

- **Unsafe low-level languages (C/C++)**

Prone to Memory Corruption Attacks

- **Hard to update**

Protection against unknown attacks

- **Predictable**

Protection overhead **on the WCET** must be predictable statically

Wanted

Robust and Predictable protection against Memory Corruption Attacks
for Real-Time Systems

Real-Time Systems - Characteristics

- **Unsafe low-level languages (C/C++)**

Prone to Memory Corruption Attacks

- **Hard to update**

Protection against unknown attacks

- **Predictable**

Protection overhead **on the WCET** must be predictable statically

Wanted

Robust and Predictable **protection against Memory Corruption Attacks**
for Real-Time Systems

Real-Time Systems - Characteristics

- **Unsafe low-level languages (C/C++)**

Prone to Memory Corruption Attacks

- **Hard to update**

Protection against unknown attacks

- **Predictable**

Protection overhead **on the WCET** must be predictable statically

Wanted

Robust and Predictable protection against Memory Corruption Attacks
for Real-Time Systems

Real-Time Systems - Characteristics

- **Unsafe low-level languages (C/C++)**

Prone to Memory Corruption Attacks

- **Hard to update**

Protection against unknown attacks

- **Predictable**

Protection overhead **on the WCET** must be predictable statically

Wanted

Robust and **Predictable** protection against Memory Corruption Attacks
for Real-Time Systems

Attacker capacity

Attacker capacity

```
char buffer[12];  
char c = 255;  
int height = getHeight();  
for(int i = 0; c != 0; i++) {  
    c = recv();  
    buffer[i] = c;  
}  
if (height >= 10000)  
    crash_plane()
```

Attacker capacity

```
char buffer[12];  
char c = 255;  
int height = getHeight();  
for(int i = 0; c != 0; i++) {  
    c = recv();  
    buffer[i] = c;  
}  
if (height >= 10000)  
    crash_plane();
```

Buffer	height	Saved PC
0	12	16
		20

Attacker capacity

```
char buffer[12];  
char c = 255;  
int height = getHeight();  
for(int i = 0; c != 0; i++) {  
    c = recv();  
    buffer[i] = c;  
}  
if (height >= 10000)  
    crash_plane();
```

Input

'A'.12+'B'.4+'C'.4

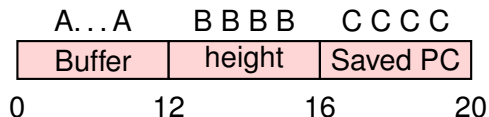
Buffer	height	Saved PC
0	12	16
		20

Attacker capacity

```
char buffer[12];  
char c = 255;  
int height = getHeight();  
for(int i = 0; c != 0; i++) {  
    c = recv();  
    buffer[i] = c;  
}  
if (height >= 10000)  
    crash_plane();
```

Input

'A'.12+'B'.4+'C'.4

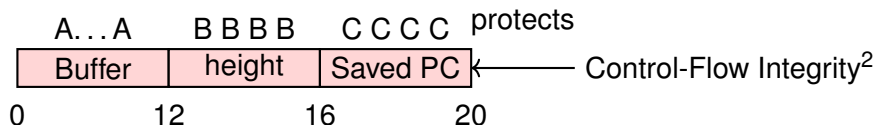


Attacker capacity

```
char buffer[12];  
char c = 255;  
int height = getHeight();  
for(int i = 0; c != 0; i++) {  
    c = recv();  
    buffer[i] = c;  
}  
if (height >= 10000)  
    crash_plane();
```

Input

'A'.12+'B'.4+'C'.4



²Control-Flow Integrity for Real-Time Embedded Systems, *Walls et al.*, ECRTS'19

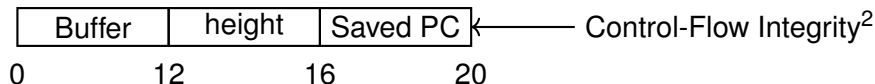
Attacker capacity

```
char buffer[12];  
char c = 255;  
int height = getHeight();  
for(int i = 0; c != 0; i++) {  
    c = recv();  
    buffer[i] = c;  
}  
if (height >= 10000)  
    crash_plane();
```

Input

'A'·12+'B'·4

protects



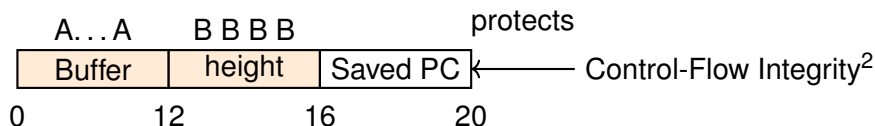
²Control-Flow Integrity for Real-Time Embedded Systems, *Walls et al.*, ECRTS'19

Attacker capacity

```
char buffer[12];  
char c = 255;  
int height = getHeight();  
for(int i = 0; c != 0; i++) {  
    c = recv();  
    buffer[i] = c;  
}  
if (height >= 10000)  
    crash_plane();
```

Input

'A'·12+'B'·4



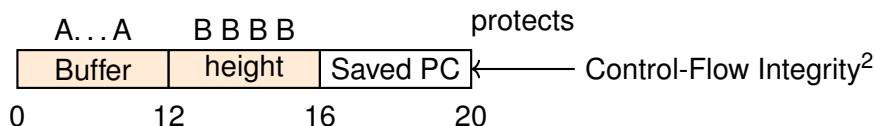
²Control-Flow Integrity for Real-Time Embedded Systems, *Walls et al.*, ECRTS'19

Attacker capacity

```
char buffer[12];  
char c = 255;  
int height = getHeight();  
for(int i = 0; c != 0; i++) {  
    c = recv();  
    buffer[i] = c;  
}  
if (height >= 10000)   
    crash_plane();
```

Input
'A'·12+'B'·4

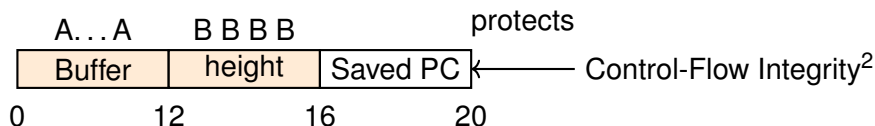
← Attacker controlled ⚠



²Control-Flow Integrity for Real-Time Embedded Systems, *Walls et al.*, ECRTS'19

Attacker capacity

```
char buffer[12];
char c = 255;
int height = getHeight();
for(int i = 0; c != 0; i++) {
    c = recv(); ← Input
    buffer[i] = c;
}
if (height >= 10000) ← Attacker controlled ⚠
    crash_plane();
```



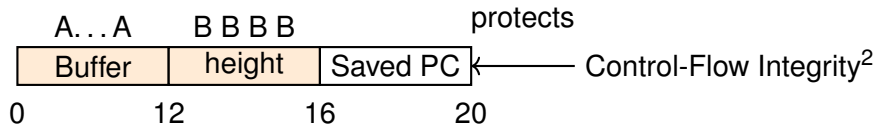
Data-Flow Attacks³

²Control-Flow Integrity for Real-Time Embedded Systems, *Walls et al.*, ECRTS'19

³Non-Control-Data Attacks Are Realistic Threats, *Chen et al.*, USENIX'05

Attacker capacity

```
char buffer[12];
char c = 255;
int height = getHeight();
for(int i = 0; c != 0; i++) {
    c = recv(); ← Input
    buffer[i] = c;
}
if (height >= 10000) ← Attacker controlled ⚠
    crash_plane();
```



Data-Flow Attacks³ $\Rightarrow$ We want to protect **all memory operations**

²Control-Flow Integrity for Real-Time Embedded Systems, *Walls et al.*, ECRTS'19

³Non-Control-Data Attacks Are Realistic Threats, *Chen et al.*, USENIX'05

Our Work : Real-Time Data-Flow Integrity

⁴ *Securing software by enforcing data-flow integrity*, Castro et al., USENIX '06.

Our Work : Real-Time Data-Flow Integrity

Data-Flow Integrity⁴ (DFI) :

⁴*Securing software by enforcing data-flow integrity*, Castro et al., USENIX '06.

Our Work : Real-Time Data-Flow Integrity

Data-Flow Integrity⁴ (DFI) : ✓ Robust

⁴*Securing software by enforcing data-flow integrity*, Castro et al., USENIX '06.

Our Work : Real-Time Data-Flow Integrity

Data-Flow Integrity⁴ (DFI) : ✓ Robust
 ✓ Static analysis **at compile time**

⁴*Securing software by enforcing data-flow integrity*, Castro et al., USENIX '06.

Our Work : Real-Time Data-Flow Integrity

Data-Flow Integrity⁴ (DFI) :

- ✓ Robust
- ✓ Static analysis **at compile time**
- ~ WCET Overhead

⁴ *Securing software by enforcing data-flow integrity*, Castro et al., USENIX '06.

Our Work : Real-Time Data-Flow Integrity

Data-Flow Integrity⁴ (DFI) : ✓ Robust
 ✓ Static analysis **at compile time**
 ~ WCET Overhead

Goal

Reduce the overhead of DFI on WCET

⁴ *Securing software by enforcing data-flow integrity*, Castro et al., USENIX '06.

Data-Flow Integrity⁴

```
p ← r1 (store)
if ( ... )
  p ← r2 (store)
else
  x ← r3 (store)
p → r4 (load)
```

⁴Securing software by enforcing data-flow integrity, Castro et al., USENIX '06.

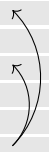
Data-Flow Integrity⁴

```
| p <- r1 (store)    Tag: 1  
| if ( ... )  
|   p <- r2 (store)  Tag: 2  
| else  
|   x <- r3 (store)  Tag: 3  
| p -> r4 (load)
```

⁴Securing software by enforcing data-flow integrity, Castro et al., USENIX '06.

Data-Flow Integrity⁴

```
| p <- r1 (store)    Tag: 1  
| if ( ... )  
|   p <- r2 (store)  Tag: 2  
| else  
|   x <- r3 (store)  Tag: 3  
| p -> r4 (load)    {1,2}
```

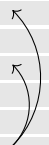


Static analysis
at compile time

⁴Securing software by enforcing data-flow integrity, Castro et al., USENIX '06.

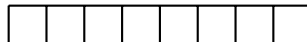
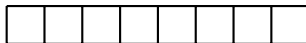
Data-Flow Integrity⁴

```
| p <- r1 (store)   Tag: 1  
| if ( ... )  
|   p <- r2 (store) Tag: 2  
| else  
|   x <- r3 (store) Tag: 3  
| p -> r4 (load)   {1,2}
```



Static analysis
at compile time

Memory



Runtime Definition Table

⁴Securing software by enforcing data-flow integrity, Castro et al., USENIX '06.

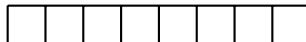
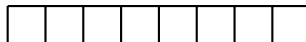
Data-Flow Integrity⁴

p <- r1 (store)	Tag: 1	
if (...)		
p <- r2 (store)	Tag: 2	
else		
x <- r3 (store)	Tag: 3	
p -> r4 (load)	{1,2}	

Static analysis
at compile time

p <- r2 (store)

Memory

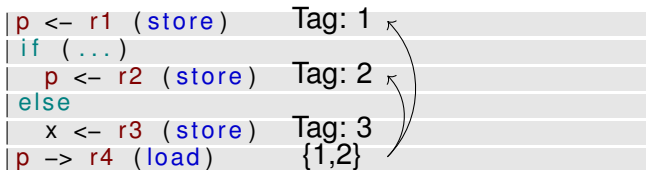


Runtime Definition Table

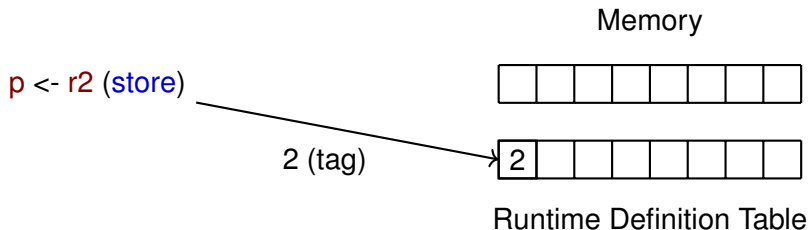
⁴Securing software by enforcing data-flow integrity, Castro et al., USENIX '06.

Data-Flow Integrity⁴

```
| p <- r1 (store)   Tag: 1  
| if ( ... )  
|   p <- r2 (store) Tag: 2  
| else  
|   x <- r3 (store) Tag: 3  
| p -> r4 (load)   {1,2}
```



Static analysis
at compile time

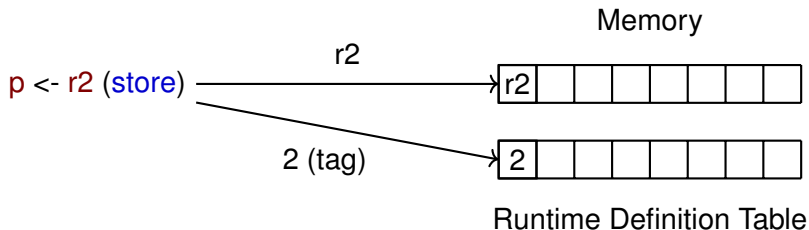


⁴Securing software by enforcing data-flow integrity, Castro et al., USENIX '06.

Data-Flow Integrity⁴

```
| p <- r1 (store)   Tag: 1  
| if ( ... )  
|   p <- r2 (store) Tag: 2  
| else  
|   x <- r3 (store) Tag: 3  
| p -> r4 (load)   {1,2}
```

Static analysis
at compile time



⁴Securing software by enforcing data-flow integrity, Castro et al., USENIX '06.

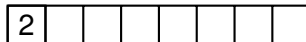
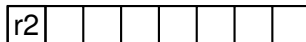
Data-Flow Integrity⁴

```
p <- r1 (store)    Tag: 1  
if ( ... )  
  p <- r2 (store)  Tag: 2  
else  
  x <- r3 (store)  Tag: 3  
p -> r4 (load)    {1,2}
```

Static analysis
at compile time

p -> r4 (load)

Memory



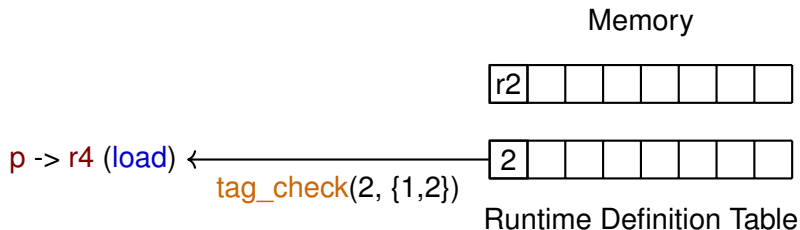
Runtime Definition Table

⁴Securing software by enforcing data-flow integrity, Castro et al., USENIX '06.

Data-Flow Integrity⁴

```
p <- r1 (store)    Tag: 1  
if ( ... )  
  p <- r2 (store)  Tag: 2  
else  
  x <- r3 (store)  Tag: 3  
p -> r4 (load)     {1,2}
```

Static analysis
at compile time

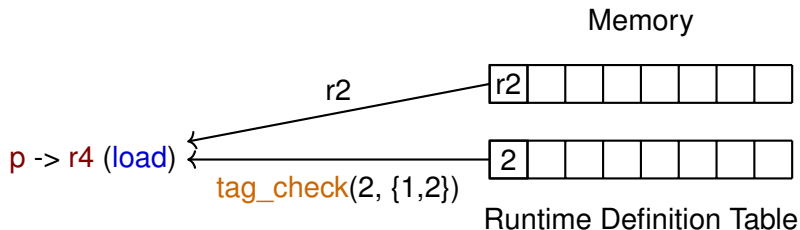


⁴Securing software by enforcing data-flow integrity, Castro et al., USENIX '06.

Data-Flow Integrity⁴

```
p <- r1 (store)    Tag: 1  
if ( ... )  
  p <- r2 (store)  Tag: 2  
else  
  x <- r3 (store)  Tag: 3  
p -> r4 (load)     {1,2}
```

Static analysis
at compile time



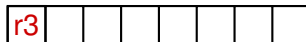
⁴Securing software by enforcing data-flow integrity, Castro et al., USENIX '06.

Data-Flow Integrity⁴

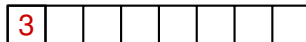
```
p <- r1 (store)    Tag: 1  
if ( ... )  
  p <- r2 (store)  Tag: 2  
else  
  x <- r3 (store)  Tag: 3  
p -> r4 (load)     {1,2}
```

Static analysis
at compile time

Memory



p -> r4 (load)



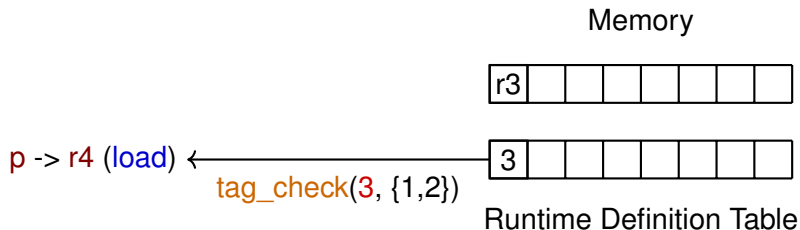
Runtime Definition Table

⁴Securing software by enforcing data-flow integrity, Castro et al., USENIX '06.

Data-Flow Integrity⁴

```
p <- r1 (store)    Tag: 1
if ( ... )
  p <- r2 (store)  Tag: 2
else
  x <- r3 (store)  Tag: 3
p -> r4 (load)     {1,2}
```

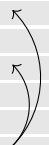
Static analysis
at compile time



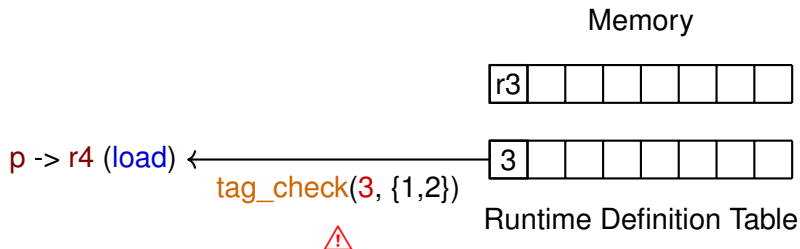
⁴Securing software by enforcing data-flow integrity, Castro et al., USENIX '06.

Data-Flow Integrity⁴

```
p <- r1 (store)    Tag: 1  
if ( ... )  
  p <- r2 (store)  Tag: 2  
else  
  x <- r3 (store)  Tag: 3  
p -> r4 (load)    {1,2}
```



Static analysis
at compile time



⁴Securing software by enforcing data-flow integrity, Castro et al., USENIX '06.

DFI⁴ - Tag check existing optimizations

```
check_tag(tag, {1, 3, 4, 5})
```

⁴Securing software by enforcing data-flow integrity, Castro et al., USENIX '06.

DFI⁴ - Tag check existing optimizations

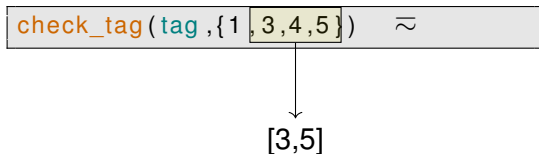
```
check_tag(tag, {1, 3, 4, 5}) ~
```

```
if tag == 1 or  
   tag == 3 or  
   tag == 4 or  
   tag == 5:  
    continue  
raise Exception()
```

4 conditions

⁴Securing software by enforcing data-flow integrity, Castro et al., USENIX '06.

DFI⁴ - Tag check existing optimizations



```
if tag == 1 or  
    tag == 3 or  
    tag == 4 or  
    tag == 5:  
    continue  
raise Exception()
```

4 conditions

⁴Securing software by enforcing data-flow integrity, Castro et al., USENIX '06.

DFI⁴ - Tag check existing optimizations

`check_tag(tag, {1, 3, 4, 5}) =`

↓

[3,5]

```
if tag == 1 or  
    3 <= tag <= 5:  
    continue  
raise Exception()
```

With intervals : 3 conditions

⁴Securing software by enforcing data-flow integrity, Castro et al., USENIX '06.

DFI⁴ - Check Tag Optimizations

⁴*Securing software by enforcing data-flow integrity*, Castro et al., USENIX '06.

DFI⁴ - Check Tag Optimizations

- Number of intervals

⁴*Securing software by enforcing data-flow integrity*, Castro et al., USENIX '06.

DFI⁴ - Check Tag Optimizations

- Number of intervals

```
if tag == 1 or  
   3 <= tag <= 4 or  
   6 <= tag <= 7:  
    continue
```

⁴Securing software by enforcing data-flow integrity, Castro et al., USENIX '06.

DFI⁴ - Check Tag Optimizations

- Number of intervals

```
if tag == 1 or  
   3 <= tag <= 4 or 7 ↦ 5  
   6 <= tag <= 7: →  
    continue
```

⁴Securing software by enforcing data-flow integrity, Castro et al., USENIX '06.

DFI⁴ - Check Tag Optimizations

- Number of intervals

```
if tag == 1 or  
  3 <= tag <= 4 or  
  6 <= tag <= 7:  
    continue
```

$7 \mapsto 5 \rightarrow$

```
if tag == 1 or  
  3 <= tag <= 6:  
    continue
```

⁴Securing software by enforcing data-flow integrity, Castro et al., USENIX '06.

DFI⁴ - Check Tag Optimizations

- Number of intervals **([4] uses a greedy algorithm)**

```
if tag == 1 or  
  3 <= tag <= 4 or  
  6 <= tag <= 7:  
    continue
```

$7 \mapsto 5 \rightarrow$

```
if tag == 1 or  
  3 <= tag <= 6:  
    continue
```

⁴Securing software by enforcing data-flow integrity, Castro et al., USENIX '06.

DFI⁴ - Check Tag Optimizations

- Number of intervals ([4] uses a greedy algorithm)

```
if tag == 1 or  
  3 <= tag <= 4 or  
  6 <= tag <= 7:  
    continue
```

$7 \mapsto 5 \rightarrow$

```
if tag == 1 or  
  3 <= tag <= 6:  
    continue
```

- Interval order

⁴Securing software by enforcing data-flow integrity, Castro et al., USENIX '06.

DFI⁴ - Check Tag Optimizations

- Number of intervals ([4] uses a greedy algorithm)

```
if tag == 1 or  
  3 <= tag <= 4 or  
  6 <= tag <= 7:  
    continue
```

$7 \mapsto 5 \rightarrow$

```
if tag == 1 or  
  3 <= tag <= 6:  
    continue
```

- Interval order

```
if tag == 1 or  
  3 <= tag <= 4:  
    continue
```

⁴Securing software by enforcing data-flow integrity, Castro et al., USENIX '06.

DFI⁴ - Check Tag Optimizations

- Number of intervals ([4] uses a greedy algorithm)

```
if tag == 1 or  
  3 <= tag <= 4 or  
  6 <= tag <= 7:  
    continue
```

$7 \mapsto 5 \rightarrow$

```
if tag == 1 or  
  3 <= tag <= 6:  
    continue
```

- Interval order

```
if tag == 1 or  
  3 <= tag <= 4:  
    continue
```

```
if 3 <= tag <= 4 or  
  tag == 1:  
    continue
```

⁴Securing software by enforcing data-flow integrity, Castro et al., USENIX '06.

DFI⁴ - Check Tag Optimizations

- Number of intervals ([4] uses a greedy algorithm)

```
if tag == 1 or  
  3 <= tag <= 4 or  
  6 <= tag <= 7:  
  continue
```

$7 \mapsto 5$

```
if tag == 1 or  
  3 <= tag <= 6:  
  continue
```

- Interval order

<pre>if tag == 1 or 3 <= tag <= 4: continue</pre>		<pre>if 3 <= tag <= 4 or tag == 1: continue</pre>
---	---	---

⁴Securing software by enforcing data-flow integrity, Castro et al., USENIX '06.

DFI⁴ - Check Tag Optimizations

- Number of intervals ([4] uses a greedy algorithm)

```
if tag == 1 or  
  3 <= tag <= 4 or  
  6 <= tag <= 7:  
  continue
```

$7 \mapsto 5$

```
if tag == 1 or  
  3 <= tag <= 6:  
  continue
```

- Interval order **(un-used by [4])**

```
if tag == 1 or  
  3 <= tag <= 4:  
  continue
```

```
if 3 <= tag <= 4 or  
  tag == 1:  
  continue
```

⁴Securing software by enforcing data-flow integrity, Castro et al., USENIX '06.

RT-DFI - Combining Everything

RT-DFI - Combining Everything

```
if tag == 1 or  
   3 <= tag <= 5:  
    continue
```

RT-DFI - Combining Everything

```
if tag == 1 or  
   3 <= tag <= 5:  
    continue
```

Tags of the context

RT-DFI - Combining Everything

```
if tag == 1 or  
   3 <= tag <= 5:  
    continue
```

Tags of the context

{ 1 }

RT-DFI - Combining Everything

Tags of the context



```
if tag == 1 or  
  3 <= tag <= 5:  
  continue
```

{ 1 }

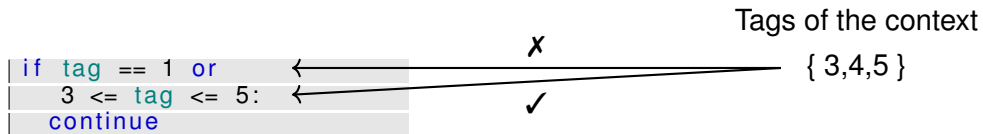
RT-DFI - Combining Everything

```
if tag == 1 or  
3 <= tag <= 5:  
    continue
```

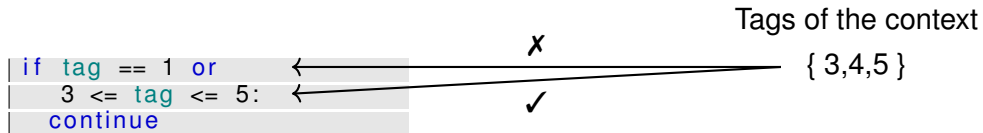
Tags of the context

{ 3,4,5 }

RT-DFI - Combining Everything

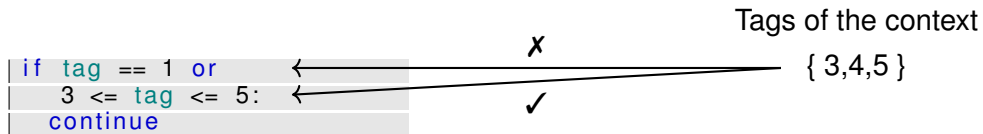


RT-DFI - Combining Everything



Possible optimizations

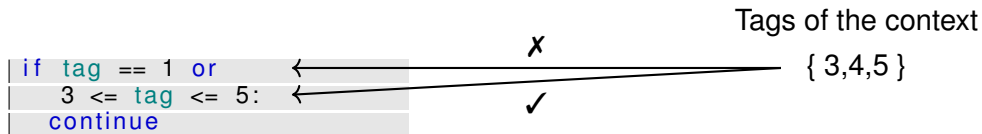
RT-DFI - Combining Everything



Possible optimizations

```
if tag == 1 or
3 <= tag <= 5:
continue
```

RT-DFI - Combining Everything

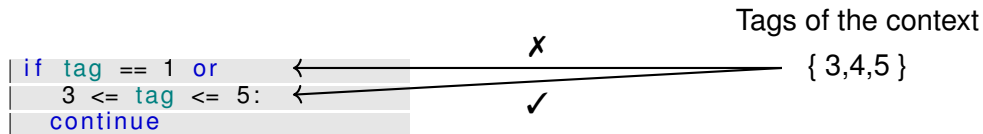


Possible optimizations

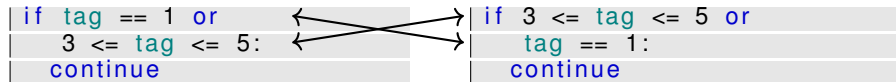
```
if tag == 1 or
    3 <= tag <= 5:
    continue
```

```
if 3 <= tag <= 5 or
    tag == 1:
    continue
```

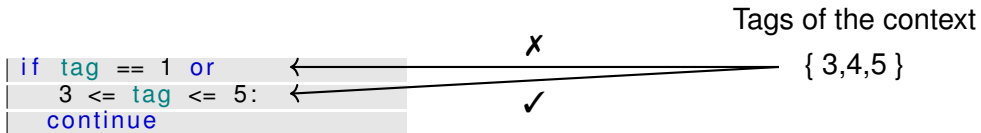
RT-DFI - Combining Everything



Possible optimizations



RT-DFI - Combining Everything

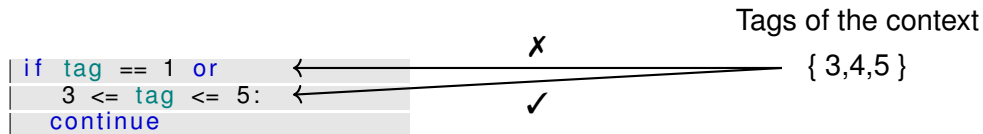


Possible optimizations

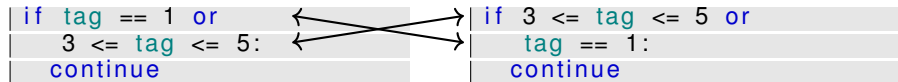


Local

RT-DFI - Combining Everything



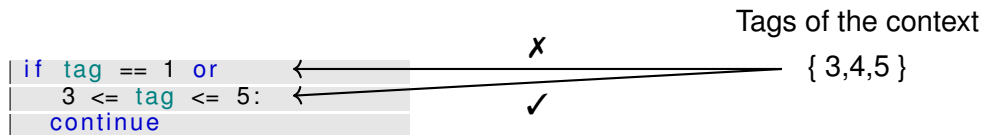
Possible optimizations



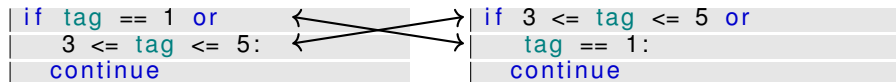
Local

{ 1,[3,5] }

RT-DFI - Combining Everything



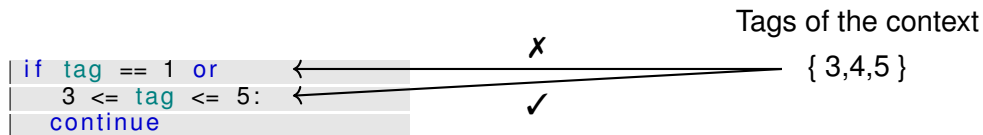
Possible optimizations



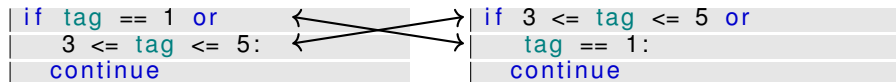
Local

$$\{ 1, [3,5] \} \xrightarrow{5 \mapsto 2}$$

RT-DFI - Combining Everything



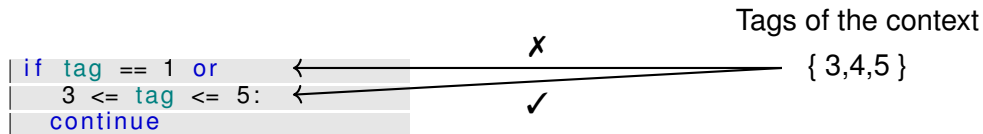
Possible optimizations



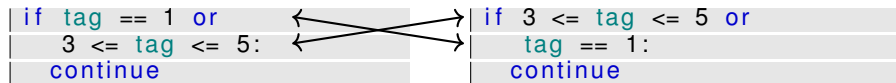
Local

$$\{ 1, [3,5] \} \xrightarrow{5 \mapsto 2} \{ [1,4] \}$$

RT-DFI - Combining Everything



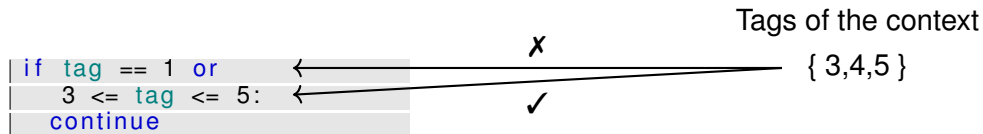
Possible optimizations



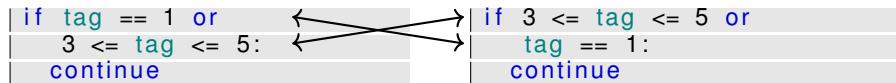
Local

$$\begin{aligned} \{ 1, [3, 5] \} &\xrightarrow{5 \mapsto 2} \{ [1, 4] \} \\ \{ [4, 5], 7 \} &\longrightarrow \{ 2, 4, 7 \} \quad \triangle! \end{aligned}$$

RT-DFI - Combining Everything



Possible optimizations

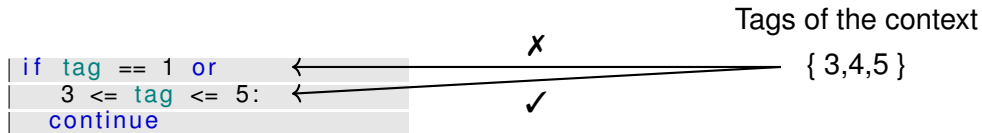


Local

$$\begin{aligned} \{ 1, [3,5] \} &\xrightarrow{5 \mapsto 2} \{ [1,4] \} \\ \{ [4,5], 7 \} &\longrightarrow \{ 2, 4, 7 \} \quad \triangle! \end{aligned}$$

Global

RT-DFI - Combining Everything



Possible optimizations



Local

$$\begin{aligned} \{ 1, [3,5] \} &\xrightarrow{5 \mapsto 2} \{ [1,4] \} \\ \{ [4,5], 7 \} &\longrightarrow \{ 2, 4, 7 \} \quad \triangle! \end{aligned}$$

Global

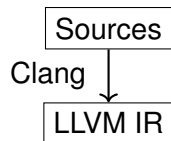
We use **Integer Linear Programming** to optimize the **DFI** on the whole **WCEP**

RT-DFI - Implementation

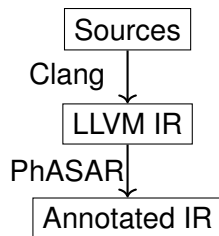
RT-DFI - Implementation

Sources

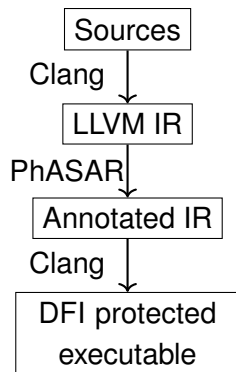
RT-DFI - Implementation



RT-DFI - Implementation

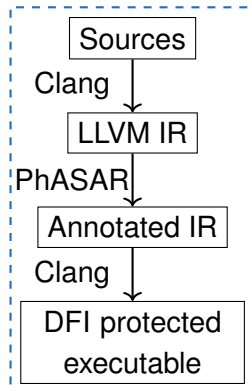


RT-DFI - Implementation

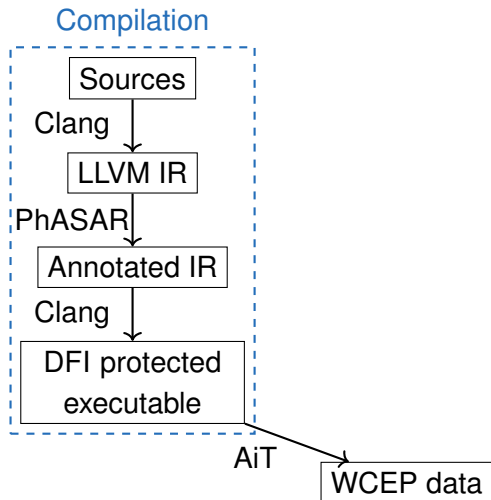


RT-DFI - Implementation

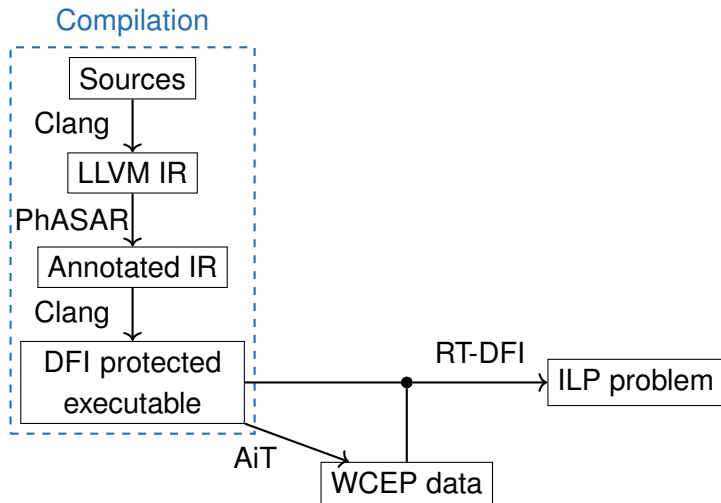
Compilation



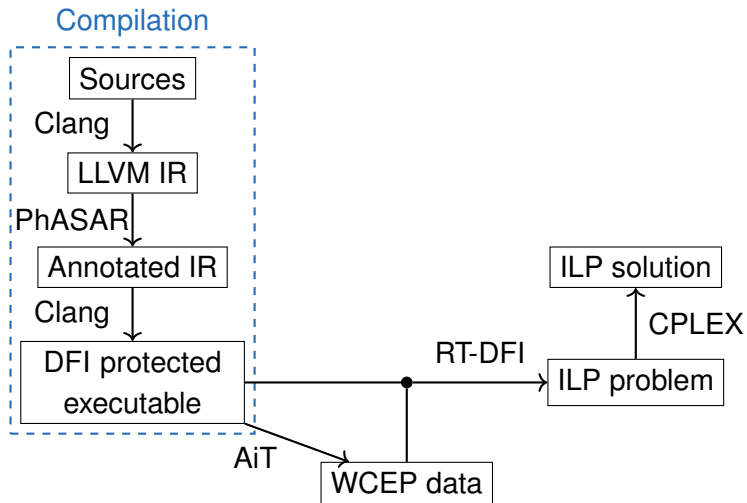
RT-DFI - Implementation



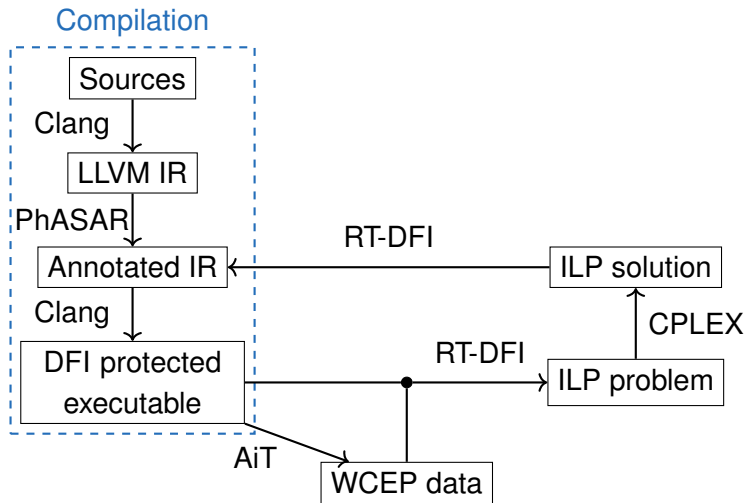
RT-DFI - Implementation



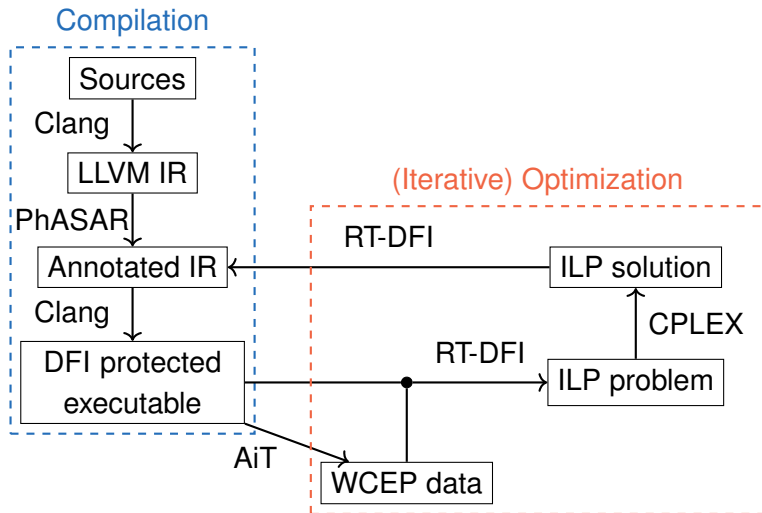
RT-DFI - Implementation



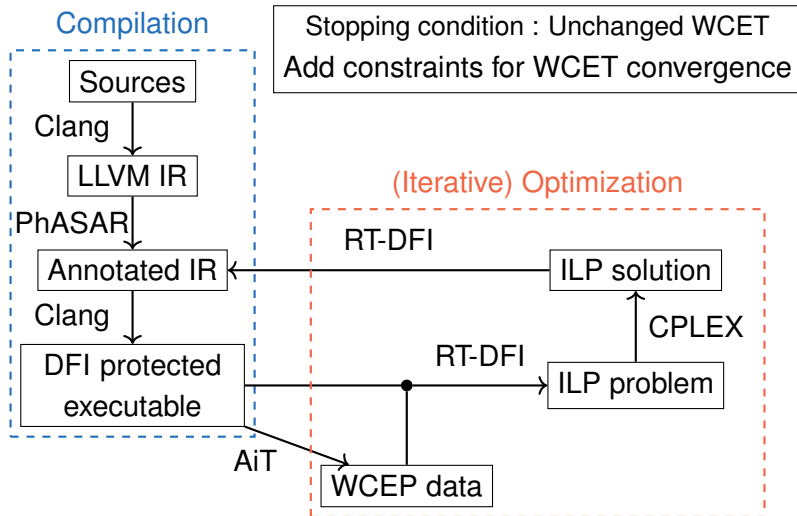
RT-DFI - Implementation



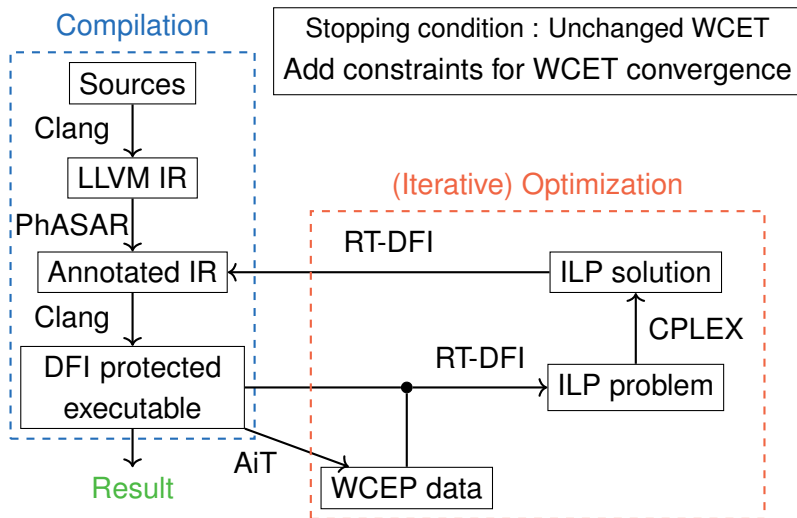
RT-DFI - Implementation



RT-DFI - Implementation



RT-DFI - Implementation

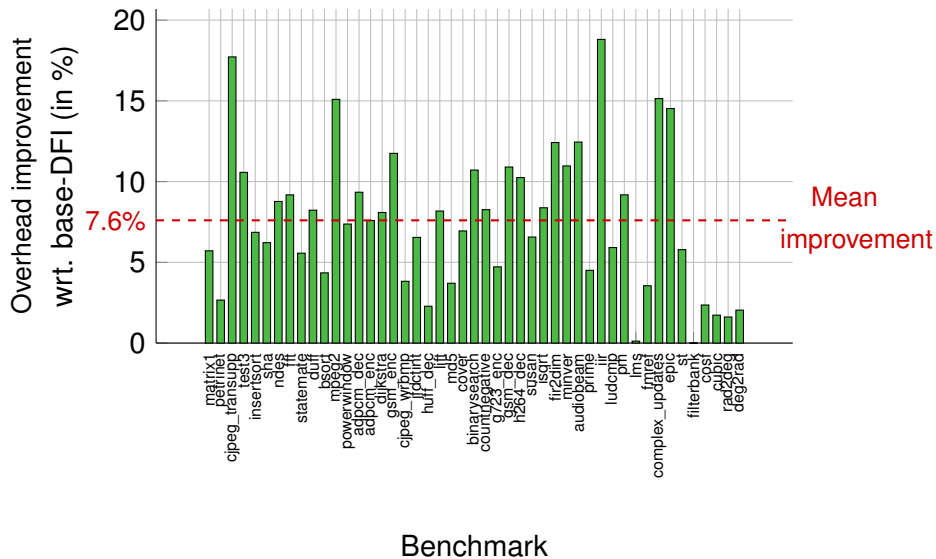


RT-DFI - Experimental Setup

- **Architecture:** RudolV (RISC-V processor)
- **WCET Estimator:** AiT (industrial standard)
- **ILP Solver:** CPLEX
- **Benchmark:** TacleBench (Real-time benchmarks, single task benches)
- **Compilation flags:** -O1
- **Stopping condition:** Unchanged WCET
- **Baseline:** Overhead of DFI (as implemented in [4]) on estimated WCET

⁴*Securing software by enforcing data-flow integrity*, Castro et al., USENIX '06.

RT-DFI - Improvement vs Baseline



Conclusion & Future Work

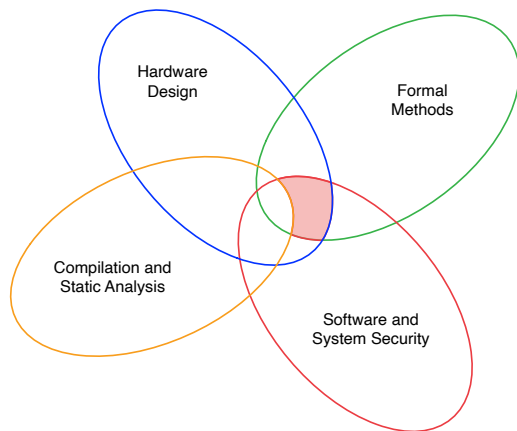
Conclusion

- Optimize DFI for the WCET (mean improvement: 7.6%)
- Iterative optimization is not efficient on the benchmarks
- See ou [ECRTS2022] paper
- Git repo: <https://gitlab.inria.fr/x-NBelle/rt-dfi>

Future Work

- Reducing tag address computation redundancy
- Handling shared resources
- WCET estimation for hardware-assisted DFI

Using Formal Methods to Reason on Hardware/Software Security

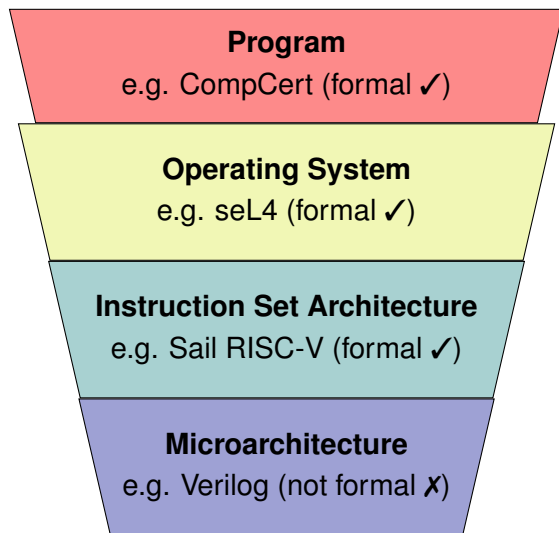


Ph.D. thesis of Matthieu Baty

Joint work with Matthieu Baty, Pierre Wilke, Alix Trieu, and Arnaud Fontaine from the ANSSI



Motivation: a stack of abstractions



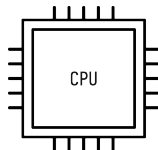
The **security of a layer** depends on the correctness of **all those below** it.

The **microarchitecture** is the lowest common denominator.

Context

We want to:

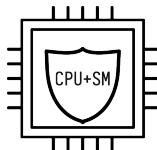
Context



We want to:

- Develop RISC-V **processors**...

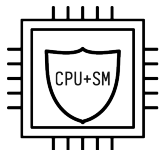
Context



We want to:

- Develop RISC-V **processors**... with **security mechanisms**!

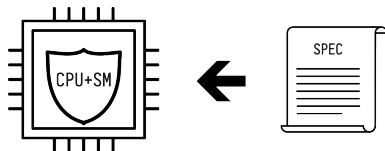
Context



We want to:

- Develop RISC-V **processors**... with **security mechanisms**!
- Describe the properties that our mechanisms enforce (**formal specification**)

Context



We want to:

- Develop RISC-V **processors**... with **security mechanisms**!
- Describe the properties that our mechanisms enforce (**formal specification**)
- Certify that our implementation is correct w.r.t. the specification (**formal proof**)

In short

We build a **formal verification framework** for a Hardware Description Language.

In short

We build a **formal verification framework** for a Hardware Description Language.

We are able to:

In short

We build a **formal verification framework** for a Hardware Description Language.

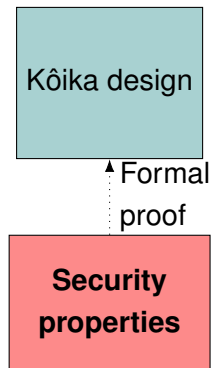


We are able to:

- Develop hardware with a formal HDL

In short

We build a **formal verification framework** for a Hardware Description Language.

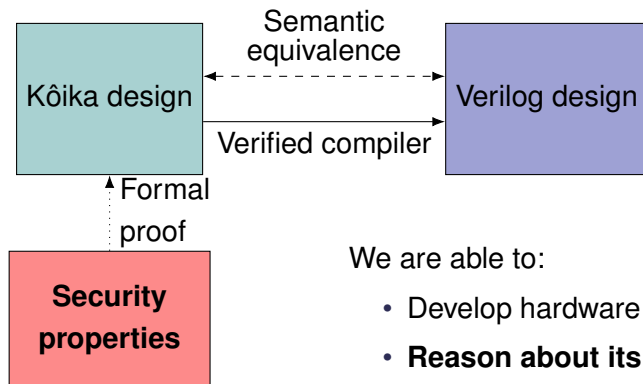


We are able to:

- Develop hardware with a formal HDL
- **Reason about its behavior**

In short

We build a **formal verification framework** for a Hardware Description Language.

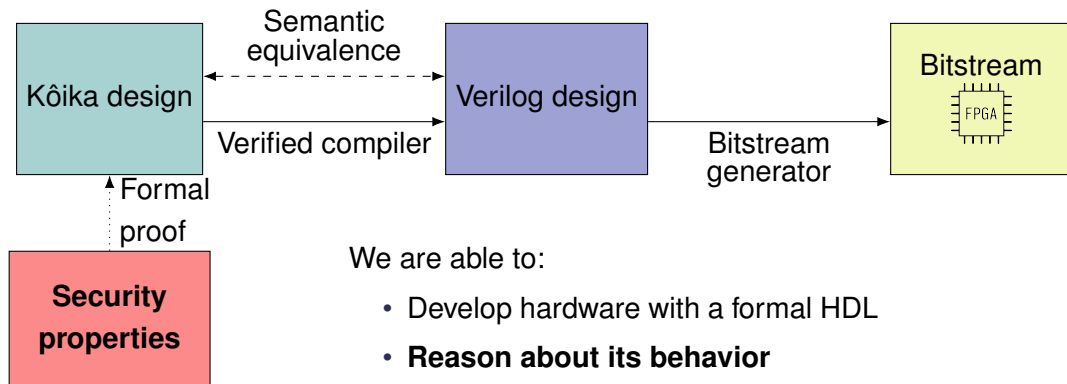


We are able to:

- Develop hardware with a formal HDL
- **Reason about its behavior**
- Simulate/synthesize it

In short

We build a **formal verification framework** for a Hardware Description Language.



We are able to:

- Develop hardware with a formal HDL
- **Reason about its behavior**
- Simulate/synthesize it

The Kôika project



The Essence of BlueSpec, PLDI'20, Thomas Bourgeat *et al.*

<https://github.com/mit-plv/koika>

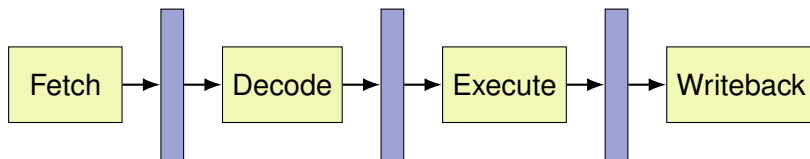
A Hardware Description Language embedded in Coq.

The Kôika HDL

Kôika is a **rule-based** register-transfer level language:

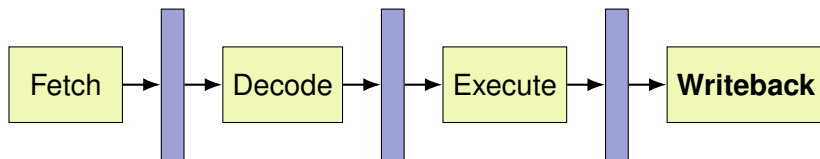
- Hardware is described as a **set of rules**
- During each cycle, all rules are scheduled to be executed
- The semantics treats the **rules sequentially** ...
- ...but the compiler ensures that everything **runs in parallel** in the end
- A rule may be **cancelled** in the presence of **conflicts**

High-level example: a pipelined processor



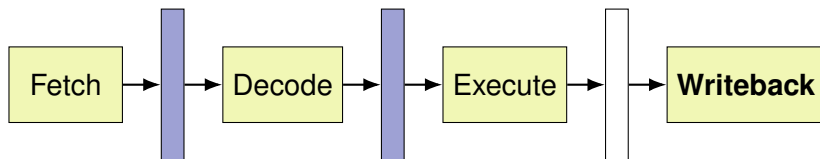
- One stage = one rule
- The stages communicate through buffers (FIFOs of size 1)
- Writing to a full FIFO is considered a conflict
- Rules are skipped on cycles on which their inclusion would lead to conflicts
- Consequence: **the stalling behavior is implicit!**

High-level example: a pipelined processor



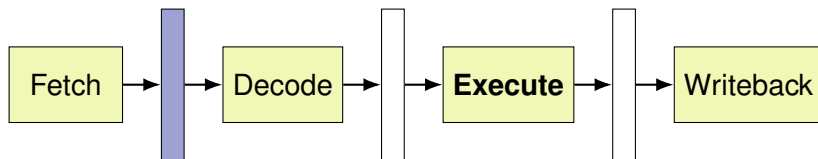
- One stage = one rule
- The stages communicate through buffers (FIFOs of size 1)
- Writing to a full FIFO is considered a conflict
- Rules are skipped on cycles on which their inclusion would lead to conflicts
- Consequence: **the stalling behavior is implicit!**

High-level example: a pipelined processor



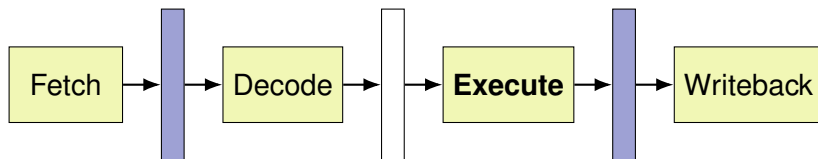
- One stage = one rule
- The stages communicate through buffers (FIFOs of size 1)
- Writing to a full FIFO is considered a conflict
- Rules are skipped on cycles on which their inclusion would lead to conflicts
- Consequence: **the stalling behavior is implicit!**

High-level example: a pipelined processor



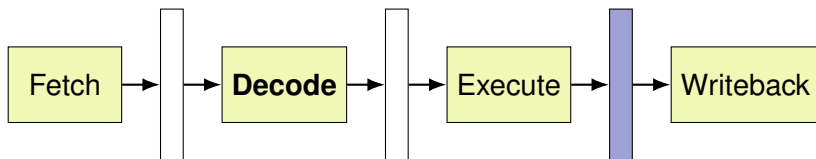
- One stage = one rule
- The stages communicate through buffers (FIFOs of size 1)
- Writing to a full FIFO is considered a conflict
- Rules are skipped on cycles on which their inclusion would lead to conflicts
- Consequence: **the stalling behavior is implicit!**

High-level example: a pipelined processor



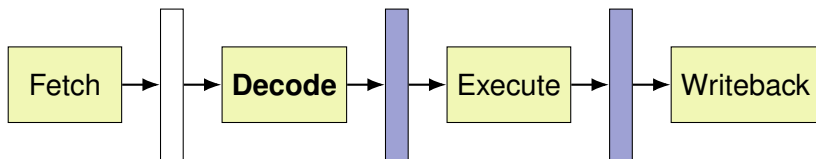
- One stage = one rule
- The stages communicate through buffers (FIFOs of size 1)
- Writing to a full FIFO is considered a conflict
- Rules are skipped on cycles on which their inclusion would lead to conflicts
- Consequence: **the stalling behavior is implicit!**

High-level example: a pipelined processor



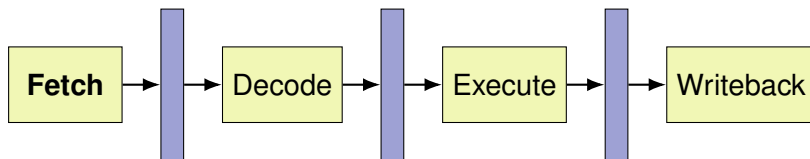
- One stage = one rule
- The stages communicate through buffers (FIFOs of size 1)
- Writing to a full FIFO is considered a conflict
- Rules are skipped on cycles on which their inclusion would lead to conflicts
- Consequence: **the stalling behavior is implicit!**

High-level example: a pipelined processor



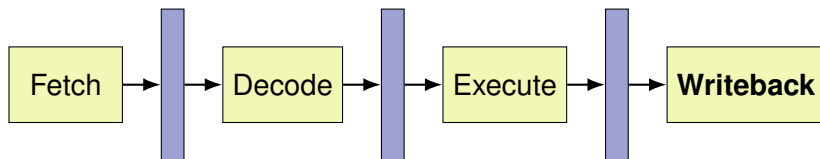
- One stage = one rule
- The stages communicate through buffers (FIFOs of size 1)
- Writing to a full FIFO is considered a conflict
- Rules are skipped on cycles on which their inclusion would lead to conflicts
- Consequence: **the stalling behavior is implicit!**

High-level example: a pipelined processor



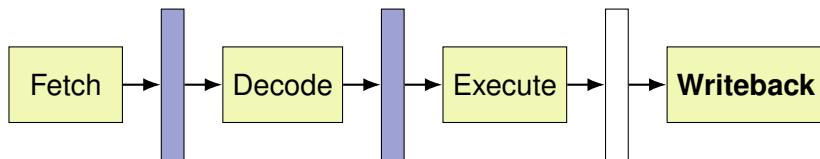
- One stage = one rule
- The stages communicate through buffers (FIFOs of size 1)
- Writing to a full FIFO is considered a conflict
- Rules are skipped on cycles on which their inclusion would lead to conflicts
- Consequence: **the stalling behavior is implicit!**

High-level example: a pipelined processor



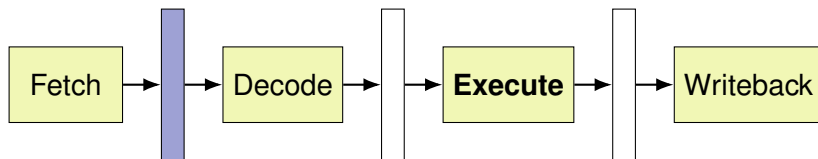
- One stage = one rule
- The stages communicate through buffers (FIFOs of size 1)
- Writing to a full FIFO is considered a conflict
- Rules are skipped on cycles on which their inclusion would lead to conflicts
- Consequence: **the stalling behavior is implicit!**

High-level example: a pipelined processor



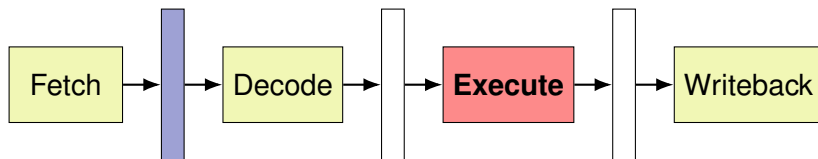
- One stage = one rule
- The stages communicate through buffers (FIFOs of size 1)
- Writing to a full FIFO is considered a conflict
- Rules are skipped on cycles on which their inclusion would lead to conflicts
- Consequence: **the stalling behavior is implicit!**

High-level example: a pipelined processor



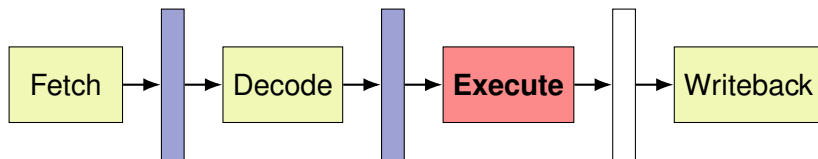
- One stage = one rule
- The stages communicate through buffers (FIFOs of size 1)
- Writing to a full FIFO is considered a conflict
- Rules are skipped on cycles on which their inclusion would lead to conflicts
- Consequence: **the stalling behavior is implicit!**

High-level example: a pipelined processor



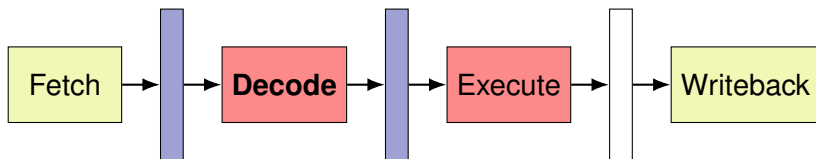
- One stage = one rule
- The stages communicate through buffers (FIFOs of size 1)
- Writing to a full FIFO is considered a conflict
- Rules are skipped on cycles on which their inclusion would lead to conflicts
- Consequence: **the stalling behavior is implicit!**

High-level example: a pipelined processor



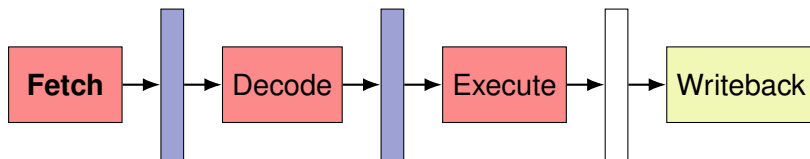
- One stage = one rule
- The stages communicate through buffers (FIFOs of size 1)
- Writing to a full FIFO is considered a conflict
- Rules are skipped on cycles on which their inclusion would lead to conflicts
- Consequence: **the stalling behavior is implicit!**

High-level example: a pipelined processor



- One stage = one rule
- The stages communicate through buffers (FIFOs of size 1)
- Writing to a full FIFO is considered a conflict
- Rules are skipped on cycles on which their inclusion would lead to conflicts
- Consequence: **the stalling behavior is implicit!**

High-level example: a pipelined processor



- One stage = one rule
- The stages communicate through buffers (FIFOs of size 1)
- Writing to a full FIFO is considered a conflict
- Rules are skipped on cycles on which their inclusion would lead to conflicts
- Consequence: **the stalling behavior is implicit!**

The processor

The Kôika developers actually provide such a processor (RISC-V):

- Embedded-class (4-stage pipeline, RV32I, unprivileged, no interrupts)
- 1000 lines of code
- Can run on FPGAs
- Not formally verified

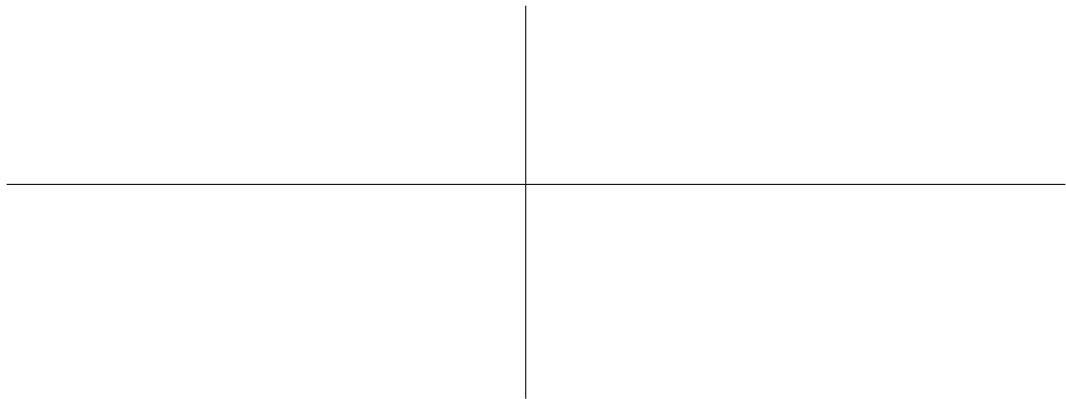
We will **extend** this processor with a **certified security mechanism**.

Shadow stack — implementation

- Our shadow stack lives in a **program-invisible** secondary memory of **limited size**
- All accesses happen **implicitly** on function calls/returns
- When a violation is detected, we **halt** the processor (remember, no interrupts)
- No support for context switching
- Less than **100 additional lines of code**

Shadow stack — specification

The properties we want to prove



Shadow stack — specification

The properties we want to prove

Shadow stack buffer **overflow**

⇒ the processor **halts**

Shadow stack — specification

The properties we want to prove

Shadow stack buffer **overflow**
 $\Rightarrow$ the processor **halts**

Shadow stack buffer **underflow**
 $\Rightarrow$ the processor **halts**

Shadow stack — specification

The properties we want to prove

Shadow stack buffer **overflow**
⇒ the processor **halts**

Shadow stack buffer **underflow**
⇒ the processor **halts**

Return to a **modified return address**
⇒ the processor **halts**

Shadow stack — specification

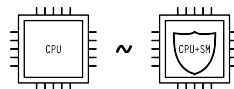
The properties we want to prove

Shadow stack buffer **overflow**
⇒ the processor **halts**

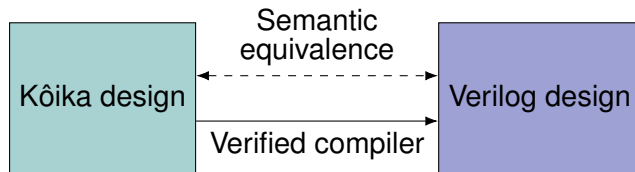
Shadow stack buffer **underflow**
⇒ the processor **halts**

Return to a **modified return address**
⇒ the processor **halts**

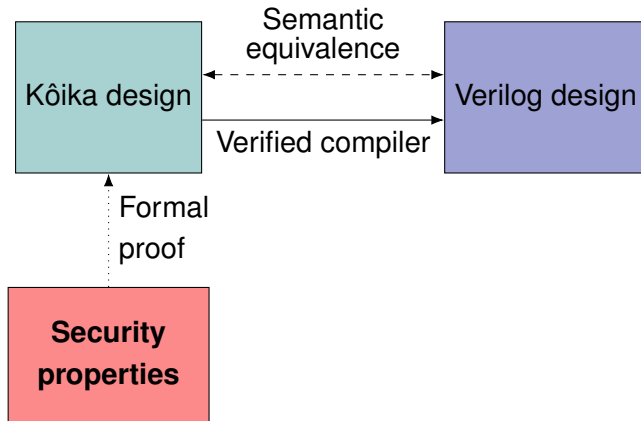
None of the above
⇒ the **behavior is preserved**



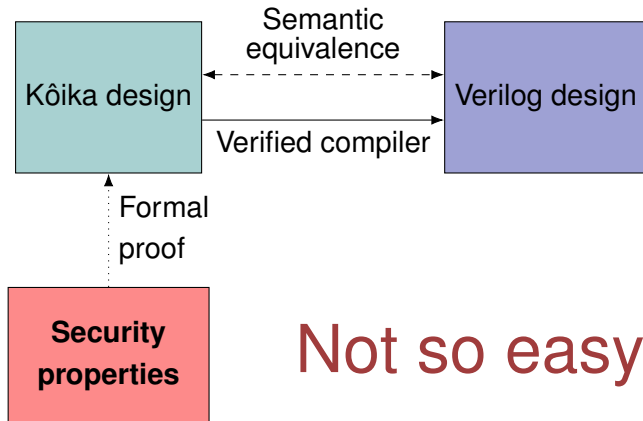
Proving properties of Kôika designs



Proving properties of Kôika designs



Proving properties of Kôika designs



Not so easy...

Problems with proofs on Kôika designs

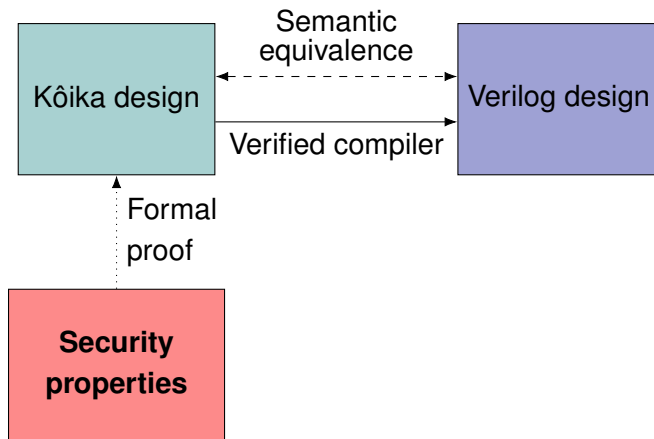
Performance! For non-trivial designs, most tactics take minutes to hours and consume an absurd amount of RAM.

No single cause, combination of:

- Rigidity of Coq's evaluation tactics
- Complexity of Kôika's semantics
- ...

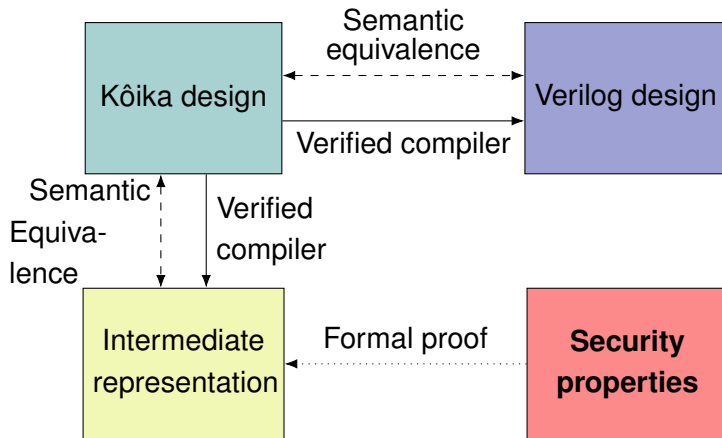
Proofs on Kôika designs

Somewhat counter-intuitively, **compiling** high-level Kôika designs into a **lower-level representation** facilitates proofs.



Proofs on Kôika designs

Somewhat counter-intuitively, **compiling** high-level Kôika designs into a **lower-level representation** facilitates proofs.



Intermediate Reasoning Representation (IRR)

IRR:

- Is a **representation of how register values are updated during a cycle.**
- Consists of:
 - A **list of variables.**
 - A **mapping from registers to the variables which describe their values** at the end of a cycle.
- **Conflicts management** is encoded **explicitly** in these expressions.

We prove the compiler from Kôika to IRR correct.

IRR isn't enough

The compiler is efficient, however it tends to produce **large sets** of deep expressions:

- Control logic is explicit
- The model is large

Furthermore, the values of the registers is usually symbolic.

Formal reasoning is still impractical. . .

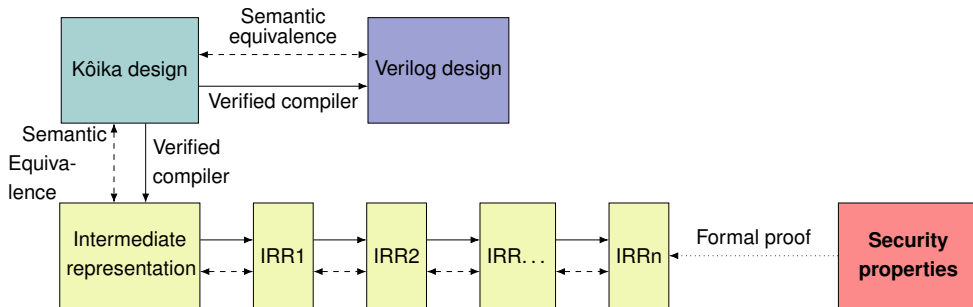
Efficient proofs with IRR

We implement a **generic collection of verified transformations** on IRRs, akin to standard compilers passes:

- Constants propagation
- Replacement of variables with an arbitrary expression proven equivalent
- Hypothesis application
- ...

These transformations can be applied **manually by the user** or using **automatic tactics**.

Conclusion



We can:

- Develop hardware with the Kôika HDL
- **Reason about its behaviour** (security properties)
- Simulate and synthesize it (the processor runs on an actual FPGA board)
- See our [CSF 2023] paper and git repo

<https://gitlab.inria.fr/explore/projects/topics/K%C3%B4ika>

Future work

- **Other security mechanisms** (e.g. memory protection, privilege levels, capabilities, enclaves)
- Security mechanisms involving **hardware/software cooperation** (e.g. CFI landing pads)
 - Ongoing work: heRVé
<https://gitlab.inria.fr/SUSHI-public/FMH/herve>
- **Functional correctness** wrt. Sail semantics
- Try to **improve modularity**, generalization of the IRR
 - Ongoing work: COQQTL, formal semantics of FIRRTL (IR of CHISEL)
- We are hiring (positions not yet advertised on the SUSHI website!)
 - 2 PhD positions
 - Two-year postdoc position

Thank you!