Protection d'un processeur embarqué RISC-V contre des attaques physiques et logicielles

Un mécanisme DIFT robuste aux injections de fautes

Vianney Lapôtre

BITFLIP by DGA

22 Novembre 2023

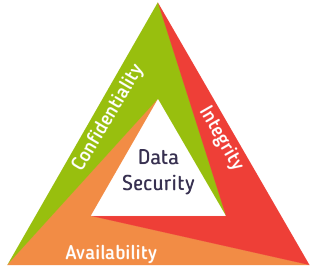
Plan
Introduction

Data security: principles

Introduction

Principles

- **Confidentiality**
- **Integrity**
- Availability



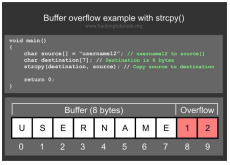
Security Policy

- Which security property is expected on each information container (file, variable, register, etc.) ?
- What operations are allowed on each container ?

Threat model

Introduction

- Software attacks: buffer overflow, ROP…



- Fault injection attacks

- Side-channel attacks not taken into account

Software security: Existing solutions

Introduction

Security mechanisms

Detect, prevent or recover from a security attack

Preventive mechanisms

Enforce the security policy:

- Cryptographic mechanisms
- Isolation (e.g., Trustzone, SAM L11)
- Formal proof, etc.

Reactive mechanisms

Monitor the system and detect any security policy violation to recover

- Intrusion detection systems (e.g., Snort, OSSEC)
  - **Dynamic Information flow tracking (DIFT)**

Dynamic Information Flow Tracking (DIFT)

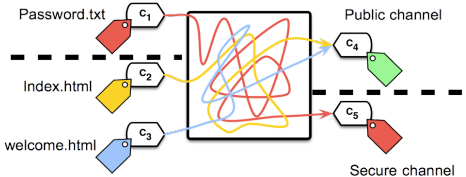
Introduction

Motivation

DIFT for security purposes : Integrity and Confidentiality

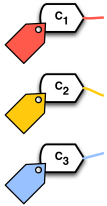
DIFT principle

- We attach **labels** called tags to **containers** and specify an information flow **policy**, i.e. relations between tags
- At runtime, we **propagate** tags to reflect information flows that occur and **detect** any **policy violation**



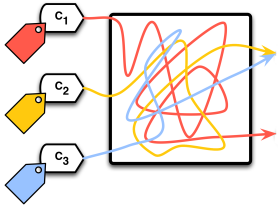
Three steps

- Tag initialization



Three steps

- Tag initialization
- Tag propagation

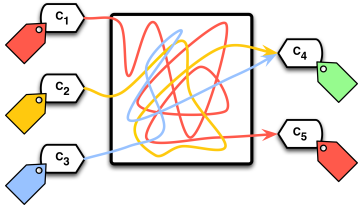


Dynamic Information Flow Tracking (DIFT)

Introduction

Three steps

- Tag initialization
- Tag propagation
- Tag check



Dynamic Information Flow Tracking (DIFT)

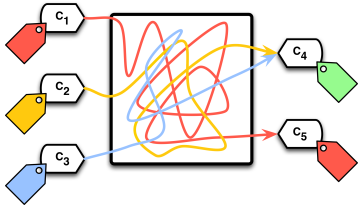
Introduction

Three steps

- Tag initialization
- Tag propagation
- Tag check

Levels of IFT

- Application level



Dynamic Information Flow Tracking (DIFT)

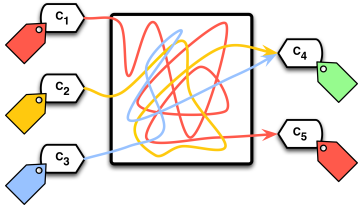
Introduction

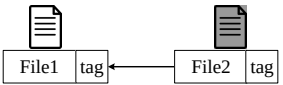
Three steps

- Tag initialization
- Tag propagation
- Tag check

Levels of IFT

- Application level
- OS level





Dynamic Information Flow Tracking (DIFT)

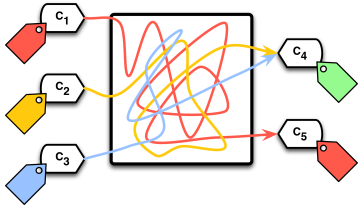
Introduction

Three steps

- Tag initialization
- Tag propagation
- Tag check

Levels of IFT

- Application level
- OS level
- Low level



OS-level Software DIFT (coarse-grained)

Introduction

Description

- Monitor is implemented within the OS kernel
- Information flows = system calls

Related Work

- Dedicated OS [4, 15, 9] : Asbestos, HiStar, Flume
- Modification of existing OS : Blare [5, 7]

Pros & Cons

+ Small runtime overhead ($< 10\%$)
+ Kernel space isolation (hardware support) helps protecting the monitor
- Overapproximation issue

Application-level Software DIFT (medium and fine-grained)

Introduction

Description

- Monitors are implemented within each application
- Information flows = affectations + conditional branching

Related Work

- Machine code [12, 6]
- Specific language [2, 11]

Pros & Cons

+ Gain in precision (hybrid analysis, SME, faceted values)
- Huge overhead ($x3$ to $x37$)
- Few or no isolation : the monitor needs to protect itself

Hardware-based DIFT (fine-grained)

Introduction

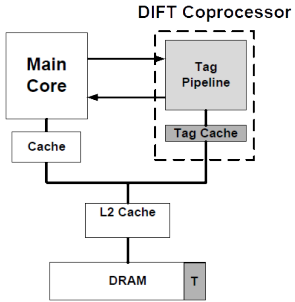


Figure: Dedicated DIFT co-processor [8, 1]

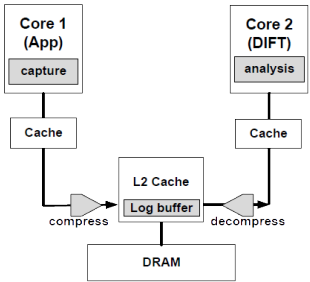


Figure: Dedicated CPU for DIFT [10]

Hardware-based DIFT (fine-grained)

Introduction

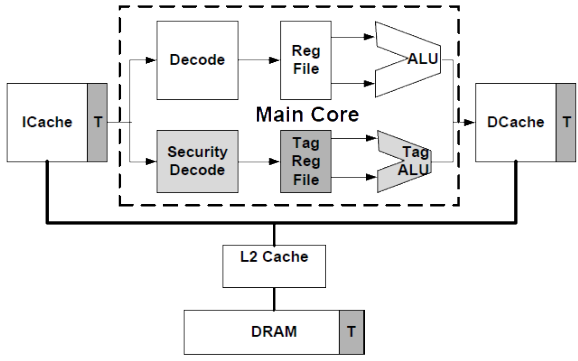


Figure: In-core DIFT [3, 13]

DIFT Example: Memory corruption

Introduction

```
int idx = tainted_input; //stdin (> BUFFER SIZE)
buffer[idx] = x; // buffer overflow
```

set r1 ← &tainted_input
load r2 ← M[r1]
add r4 ← r2 + r3
store M[r4] ← r5

pseudo-code

T	Register file
	r1
	r2
	r3:&buffer
	r4
	r5:x

T	Data memory
	Return Address
	int buffer[Size]

DIFT Example: Memory corruption

Introduction

```
int idx = tainted_input; //stdin (> BUFFER SIZE)
buffer[idx] = x; // buffer overflow
```

set r1 ← &tainted_input
load r2 ← M[r1]
add r4 ← r2 + r3
store M[r4] ← r5

pseudo-code

T	Register file
	r1:&input
	r2
	r3:&buffer
	r4
	r5:x

T	Data memory
	Return Address
	int buffer[Size]

DIFT Example: Memory corruption

Introduction

```
int idx = tainted_input; //stdin (> BUFFER SIZE)
buffer[idx] = x; // buffer overflow
```

set r1 ← &tainted_input
load r2 ← M[r1]
add r4 ← r2 + r3
store M[r4] ← r5

pseudo-code

T	Register file
	r1:&input
	r2:idx=input
	r3:&buffer
	r4
	r5:x

T	Data memory
	Return Address
	int buffer[Size]

DIFT Example: Memory corruption

Introduction

```
int idx = tainted_input; //stdin (> BUFFER SIZE)
buffer[idx] = x; // buffer overflow
```

set r1 ← &tainted_input
load r2 ← M[r1]
add r4 ← r2 + r3
store M[r4] ← r5

pseudo-code

T	Register file
	r1:&input
	r2:idx=input
	r3:&buffer
	r4:&buffer+idx
	r5:x

T	Data memory
	Return Address
	int buffer[Size]

DIFT Example: Memory corruption

Introduction

```
int idx = tainted_input; //stdin (> BUFFER SIZE)
buffer[idx] = x; // buffer overflow
```

set r1 ← &tainted_input
load r2 ← M[r1]
add r4 ← r2 + r3
store M[r4] ← r5

pseudo-code

T	Register file
	r1:&input
	r2:idx=input
	r3:&buffer
	r4:&buffer+idx
	r5:x

T	Data memory
	Return Address
	int buffer[Size]

Plan

D-RI5CY processor

- Introduction
- D-RI5CY processor
- Fault Injection Attacks against D-RI5CY
- Conclusions

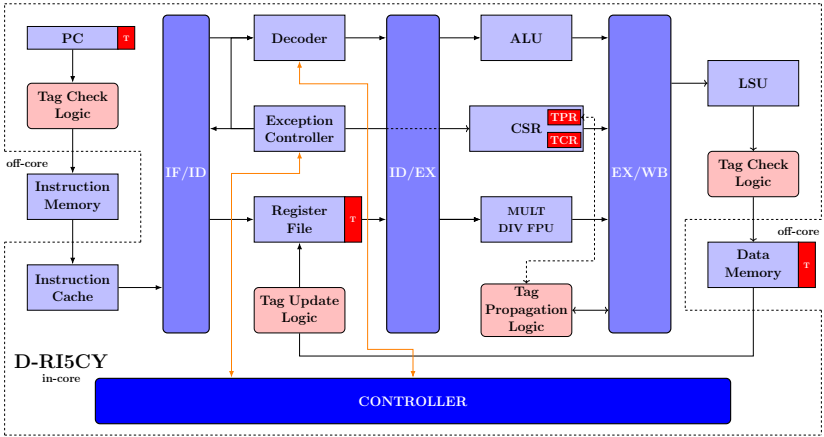
Overview

D-RI5CY processor

- Fork of the RI5CY processor [14]
  - 4-stage in-order 32-bit RISC-V optimized for low-power embedded systems and IoT application
  - Fully supports the base integer instruction set (RV32I), compressed instructions (RV32C) and the multiplication instruction set extension (RV32M) of the RISC-V ISA. In addition, it implements a set of custom extensions (RV32XPulp)
- *The D-RI5CY must be able to detect and stop various known memory-corruption attacks; the protection must be flexible and extendable through software programmable security policies to target future kinds of attacks; finally, the protection must provide a transparent and fine-grain management of security with no latency and small storage overhead*

Block diagram

D-RI5CY processor



- In red and pink the DIFT components

Tag initialization

D-RI5CY processor

- To initialize the security tags of user-supplied inputs to one, four new instructions have been implemented
  - **p.set rd** sets to one the security tag of the destination register `rd`;
  - **p.spsb xo, offset(rt)** sets to one the security tag of the memory byte at the address `rt + offset`;
  - **p.spsh xo, offset(rt)** sets to one the security tags of the memory half-word at the address `rt + offset`;
  - **p.spsw xo, offset(rt)** sets to one the security tags of the memory word at the address `rt + offset`.

Tag Propagation

D-RI5CY processor

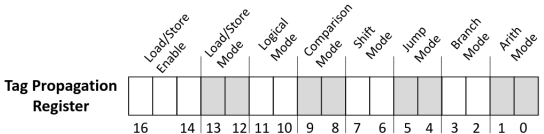


Figure: D-RI5CY Tag Propagation Register [13]

- A Mode field for each class of instructions which specifies how to propagate the tags of the input operands to the output operand tag.
  - the output tag keeps its old value (00);
  - the output tag is set to one, if both the input tags are set to one (01);
  - the output tag is set to one, if at least one input tag is set to one (10);
  - the output tag is set to zero (11).
- The three bits in the L/S enable field allow the policy to enable the source, source-address, and destination-address tags, respectively

Tag Propagation

D-RI5CY processor

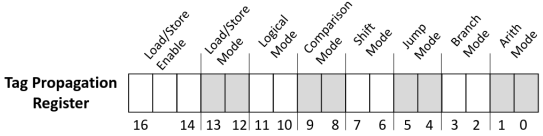


Figure: D-RI5CY Tag Propagation Register [13]

Field	Value	Rule
Load/Store Enable	001	Source tag enabled
Load/Store Mode	10	Dest tag = Source tag
Logical Mode	10	Dest tag = Source1 tag OR Source2 tag
Comparison Mode	00	No Propagation
Shift Mode	10	Dest tag = Source1 tag OR Source2 tag
Jump Mode	10	JAL: New PC = Old PC JALR: New PC = Source tag
Branch Mode	00	No propagation
Integer Arith Mode	10	Dest tag = Source1 tag OR Source2 tag

Figure: Tag Propagation Register configuration example [13]

Tag Checking

D-RI5CY processor

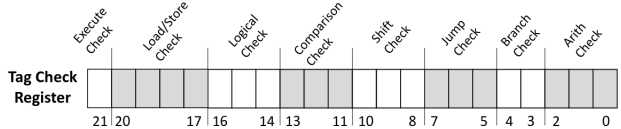


Figure: D-RI5CY Tag Check Register [13]

- The tag-check rules restrict the operations that may be performed on tagged data. If the check bit for an operand tag is set to one and the corresponding tag is equal to one, an exception is raised.
  - For all the classes except Load/Store, there are three tags to consider: first input, second input, and output tags
  - For the Load/Store class there are four tags to take into account: source-address, source, destination-address, and destination tags
  - the additional Execute Check field is associated with the program counter and specifies whether to raise a security exception when the program-counter tag is set to one

Tag Checking

D-RI5CY processor

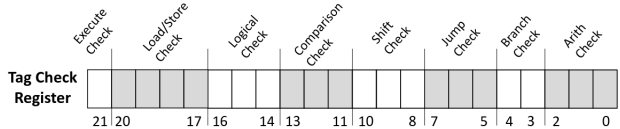


Figure: D-RI5CY Tag Check Register [13]

FIELD	VALUE	RULE
Load/Store Check	1010	Source address tag checked Destination address tag checked
Logical Check	000	No check
Comparison Check	011	Source1 tag checked Source2 tag checked
Shift Check	000	No check
Jump Check	000	No check
Branch Check	00	No check
Integer Arith Check	000	No check
Execute Check	1	Program Counter checked

Figure: Tag Check Register configuration example [13]

Plan

Fault Injection Attacks against D-RI5CY

Motivation and fault model

Fault Injection Attacks against D-RI5CY

- Identify vulnerabilities of the D-RI5CY DIFT mechanism when considering FIA and propose countermeasures
- We consider an attacker able to
  - combine software and physical attacks to defeat the DIFT mechanism
  - inject faults in registers associated to the DIFT-related components
    - set to 0, set to 1, or a bit-flip at a random position of the targeted register
- In this presentation, we consider 2 use cases: buffer overflow and Format string attacks

Buffer overflow attack

Fault Injection Attacks against D-RI5CY

- The attacker exploits a buffer overflow to reach the return address (ra) register

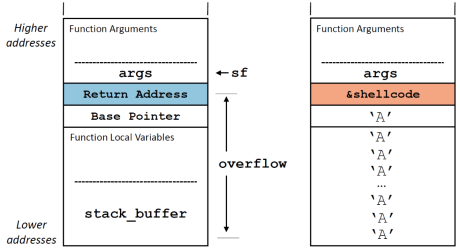


illustration from [13]

Buffer overflow attack

Fault Injection Attacks against D-RI5CY

- The attacker exploits a buffer overflow to reach the return address (ra) register

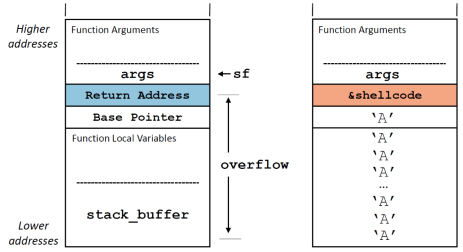
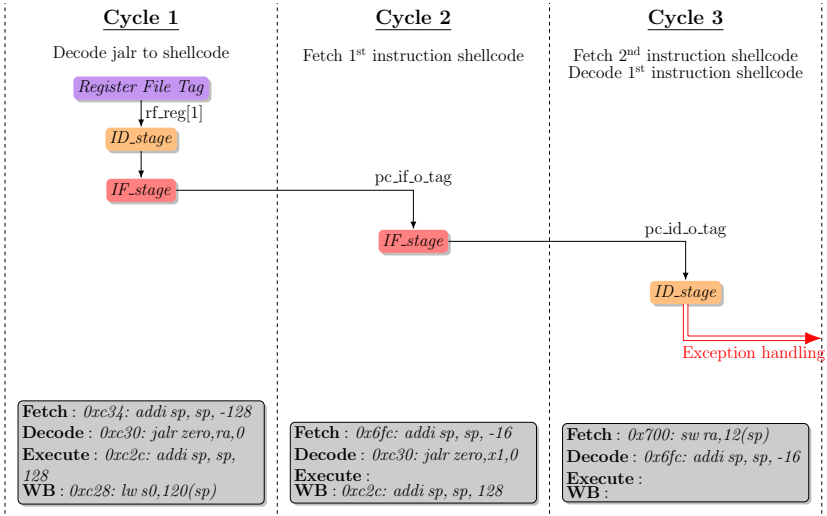


illustration from [13]

- Due to the DIFT mechanism, the tag associated with the buffer data overwrites the `ra` register tag.
- Since the buffer data is manipulated by the user, it is tagged as *not trusted*.
- When returning from the called function, the corrupted `ra` register is loaded into `PC` via a `jalr` instruction.

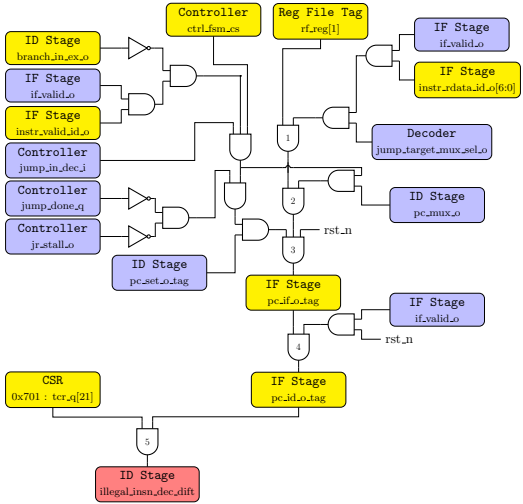
Tag propagation in a buffer overflow attack

Fault Injection Attacks against D-RI5CY



Tag propagation in a buffer overflow attack - logic view

Fault Injection Attacks against D-RI5CY

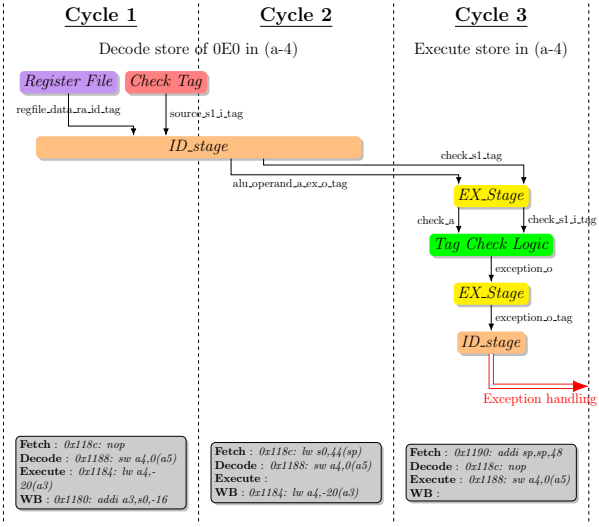


Format string attack

- The vulnerability is the use of an unchecked user input as the format string parameter in functions that perform formatting, e.g. printf()
- An attacker can use the format tokens, to write into arbitrary locations of memory, e.g. the return address of the function.
- We consider the example of Format string attack avaible at `https://github.com/sld-columbia/riscv-dift/tree/master/pulpino_apps_dift/wu-ftpd`

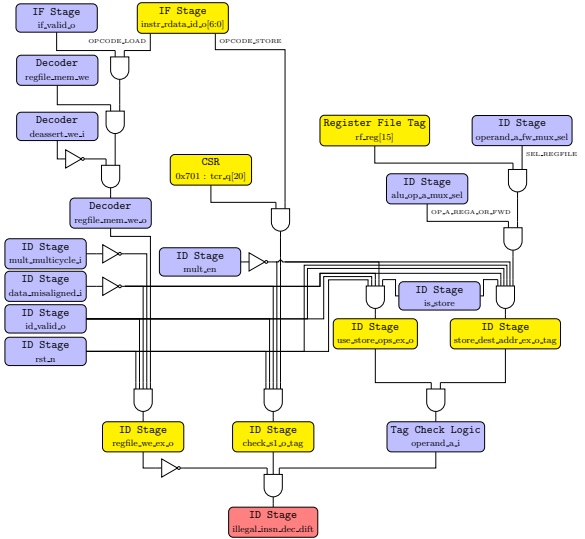
Tag propagation in a Format string attack

Fault Injection Attacks against D-RI5CY



Tag propagation in a Format string attack - logic view

Fault Injection Attacks against D-RI5CY



fault simulation campaign

Fault Injection Attacks against D-RI5CY

- Logical fault injection simulation is used for preliminary evaluations
  - faults are injected in the HDL code at cycle accurate and bit accurate level
  - a set of 54 DIFT-related registers are targeted
  - a set of attack windows are determined based on the previous study
  - set to 0, set to 1, or a bit-flip at a random position are considered
  - results are classed in four groups
    - crash: reference cycle count exceeded,
    - nothing Significant To Report (NSTR)
    - delay: illegal instruction is delayed
    - success: DIFT has been bypassed

fault simulation campaign - target registers

Table: DIFT-related registers number and size

Name	Number of registers	Size of registers (in bits)
FETCH stage	2	2
DECODE stage	13	18
RF TAG	32	32
EXECUTION stage	1	1
CSR	2	64
LSU	4	9
TOTAL	54	126

fault simulation campaign - main results

Fault Injection Attacks against D-RI5CY

Table: Fault simulations end status

	Crash	NSTR	Delay	Success	Total
Buffer overflow	0	940	17	15	972
Format string	0	1036	69	29	1134

fault simulation campaign - Buffer overflow

Fault Injection Attacks against D-RI5CY

Table: Buffer overflow: success per register, fault type and simulation time

	137140 ns		137180 ns			137220 ns		137260 ns		137300 ns
	set to 0	set to 1	set to 0	set to 1	bitflip	set to 0	bitflip	set to 0	bitflip	set to 0
pc_if_o_tag						✓		✓	✓	
rf_reg[1]						✓	✓			
tcr_q	✓		✓		✓	✓		✓		✓
tpr_q	✓	✓	✓	✓						

fault simulation campaign - format string

Fault Injection Attacks against D-RI5CY

Table: Format string attack: success per register, fault type and simulation time

	2099140 ns			2099180 ns		2099220 ns		2099260 ns		2099300 ns		2099340 ns		2099380 ns	
	set to 0	set to 1	bitflip	set to 0	set to 1	set to 0	set to 1	set to 0	set to 1	set to 0	bitflip	set to 0	bitflip	set to 0	bitflip
alu_operand_b_ex_o_tag	✓		✓												
alu_operator_o_mode	✓	✓	✓												
check_s1_o_tag												✓		✓	✓
store_dest_addr_ex_o_tag												✓		✓	✓
use_store_ops_ex_o												✓		✓	✓
rf_reg[15]										✓	✓	✓	✓		
tcr_q	✓			✓		✓		✓		✓		✓			
tpr_q		✓	✓		✓		✓		✓						

Plan
Conclusions

- We have shown that the D-RI5CY DIFT mechanism is vulnerable to FIAs
- We identified 12 DIFT-related sensitive registers
- 72 simulated fault injections over 3726 have lead to a successful attack (1.93%)

- We have shown that the D-RI5CY DIFT mechanism is vulnerable to FIAs
- We identified 12 DIFT-related sensitive registers
- 72 simulated fault injections over 3726 have lead to a successful attack (1.93%)
- In future works we will
  - Strengthen the proposed analysis through actual fault injection campaign targeting a FPGA implementation
  - Propose a robust in-core DIFT mechanism against FIAs

Protection d’un processeur embarqué RISC-V contre des attaques physiques et logicielles

Many thanks to William Pensec for his work
Thank you for listening!
Any questions?

[1] Abdul Wahab, M., Cotret, P., Nasr Allah, M., Hiet, G., Lapotre, V., and Gogniat, G.
Towards a hardware-assisted information flow tracking ecosystem for ARM processors.
In *26th International Conference on Field-Programmable Logic and Applications (FPL 2016)* (Lausanne, Switzerland, Aug. 2016).

[2] Chandra, D., and Franz, M.
Fine-grained information flow analysis and enforcement in a java virtual machine.
In *ACSAC* (2007), pp. 463–475.

[3] Dalton, M., Kannan, H., and Kozyrakis, C.
Raksha: A flexible information flow architecture for software security.
SIGARCH Comput. Archit. News. 35, 2 (June 2007), 482–493.

References

Conclusions

[4] Efstathopoulos, P., Krohn, M., VanDeBogart, S., Frey, C., Ziegler, D., Kohler, E., Mazières, D., Kaashoek, F., and Morris, R.
Labels and event processes in the asbestos operating system.
In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles* (New York, NY, USA, 2005), ACM, pp. 17–30.

[5] Geller, S., Hauser, C., Tronel, F., and Tong, V. V. T.
Information flow control for intrusion detection derived from mac policy.
In *IEEE International Conference on Communications* (2011).

[6] Harris, W. R., Jha, S., and Reps, T.
Difc programs by automatic instrumentation.
In *CCS '10: Proceedings of the 17th ACM conference on Computer and communications security* (New York, NY, USA, 2010), ACM, pp. 284–296.

References

Conclusions

[7] Hauser, C., Tronel, F., Reid, J., and Fidge, C.
A taint marking approach to confidentiality violation detection.
In *Australasian Information Security Conference (AISC 2012)* (Melbourne, Australia, 2012), C. Pieprzyk, J.and Thomborson, Ed., vol. 125 of *CRPIT*, ACS, pp. 83–90.

[8] Kannan, H., Dalton, M., and Kozyrakis, C.
Decoupling dynamic information flow tracking with a dedicated coprocessor.
In *Dependable Systems & Networks, 2009.* (2009), IEEE, pp. 105–114.

[9] Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M. F., Kohler, E., and Morris, R.
Information flow control for standard os abstractions.
In *Proceedings of the 21st Symposium on Operating Systems Principles* (Stevenson, WA, October 2007).

References

Conclusions

[10] Nagarajan, V., Kim, H.-S., Wu, Y., and Gupta, R.
Dynamic information tracking on multicores.
In *INTERACT* (Feb 2008).

[11] Nair, S. K., Simpson, P. N. D., Crispo, B., and Tanenbaum, A. S.
A virtual machine based information flow control system for policy enforcement.
In *First International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM 2007)* (Dresden, Germany, 2007), pp. 1–11.

[12] Newsome, J., and Song, D.
Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software.
In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS 2005)* (San Diego, CA, February 2005).

[13] Palmiero, C., Di Guglielmo, G., Lavagno, L., and Carloni, L. P.
Design and Implementation of a Dynamic Information Flow Tracking Architecture to Secure a RISC-V Core for IoT Applications.
In *High Performance Extreme Computing* (2018).

[14] Traber, A., Gautsch, M., and Davide, S. P.
RI5CY: User Manual Revision 1.7.
ETH Zurich, University of Bologna, 2017.

[15] Zeldovich, N., Boyd-Wickizer, S., Kohler, E., and Mazières, D.
Making information flow explicit in histar.
In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 263–278.