HardBlare, a hardware/software co-design approach for Information Flow Control

Guillaume Hiet [α], Pascal Cotret [β]

[α] CentraleSupélec, guillaume.hiet@centralesupelec.fr
[β] R&D engineer, pascal.cotret@gmail.com

June 22, 2018

HardBlare project

Started in October 2015.

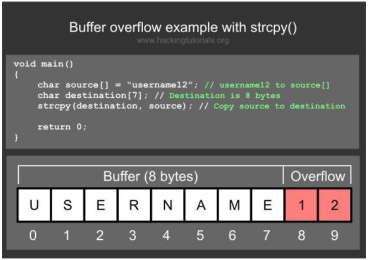
Partners (all from Brittany !)

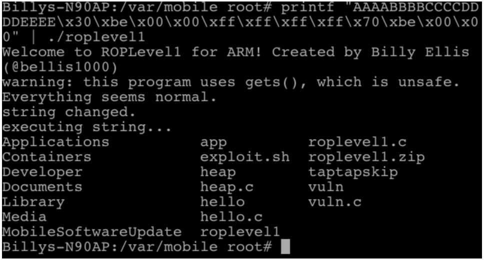
- IETR/CentraleSupélec (SCEE) @ Rennes
  - Pascal Cotret ~~(Ass. Prof.)~~ now engineer at Thales
  - Muhammad Abdul Wahab (PhD student)
- IRISA/CentraleSupélec/Inria (CIDRE) @ Rennes
  - Guillaume Hiet (Ass. Prof.)
  - Mounir Nasr Allah (PhD student)
- Lab-STICC/UBS @ Lorient
  - Guy Gogniat (Full Prof.), Vianney Lapôtre (Ass. Prof.)
  - Arnab Kumar Biswas (Postdoc)

What do we do in this project ?

Hardware extensions for DIFT/DIFC (**Dynamic Information Flow Tracking / Dynamic Information Flow Control**) on embedded processors

Threat model





- Side-channel attacks not taken into account
- Software attacks: buffer overflow, ROP...

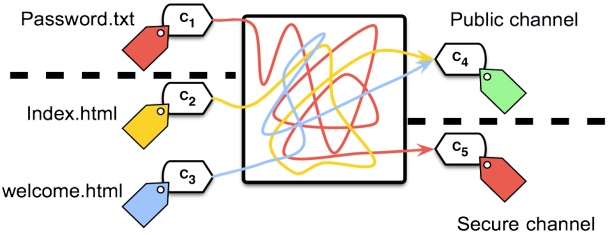
Dynamic Information Flow Tracking

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**



DIFT Example: Memory corruption

Attacker overwrites return address and takes control

```
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	Data
	r1
	r2
	r3:&buffer
	r4
	r5:x

T	Data
	Return Address
	int buffer[Size]

DIFT Example: Memory corruption

Attacker overwrites return address and takes control

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

set r1 $\leftarrow$ &tainted_input
load r2 $\leftarrow$ M[r1]
add r4 $\leftarrow$ r2 + r3
store M[r4] $\leftarrow$ r5

pseudo-code

T	Data
	r1:&input
	r2
	r3:&buffer
	r4
	r5:x

T	Data
	Return Address
	int buffer[Size]

DIFT Example: Memory corruption

Attacker overwrites return address and takes control

```
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	Data
(red)	`r1:&input`
(red)	`r2:idx=input`
(green)	`r3:&buffer`
(green)	`r4`
(green)	`r5:x`

T	Data
(green)	Return Address
(green)	int buffer[Size]

DIFT Example: Memory corruption

Attacker overwrites return address and takes control

```
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	Data
	r1:&input
	r2:idx=input
	r3:&buffer
	r4
	r5:x

T	Data
	Return Address
	int buffer[Size]

DIFT Example: Memory corruption

Attacker overwrites return address and takes control

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

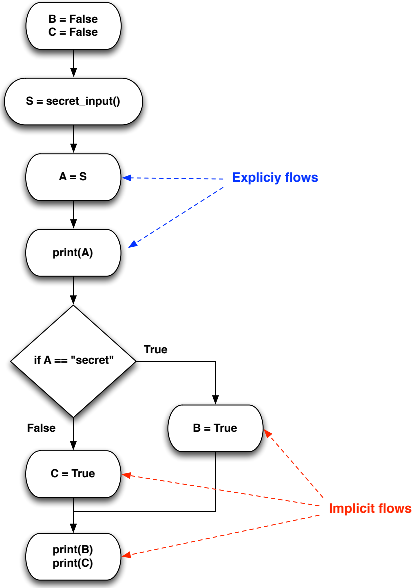
set r1 $\leftarrow$ &tainted_input
load r2 $\leftarrow$ M[r1]
add r4 $\leftarrow$ r2 + r3
store M[r4] $\leftarrow$ r5

pseudo-code

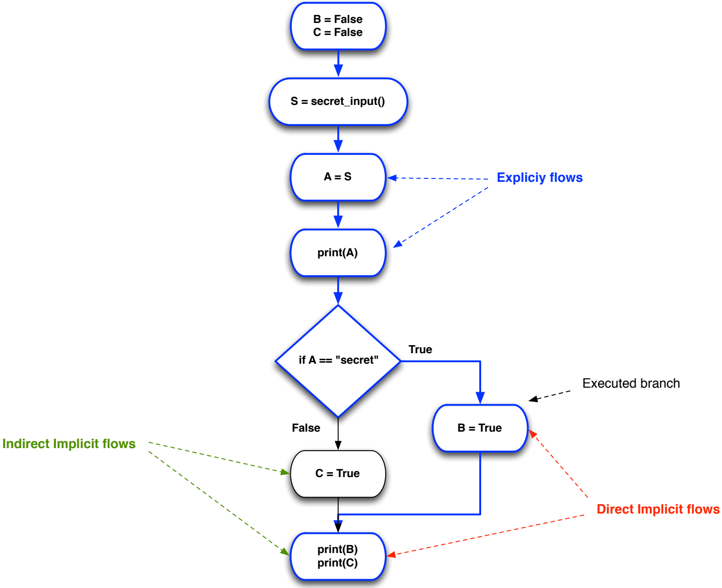
T	Data
	`r1:&input`
	`r2:idx=input`
	`r3:&buffer`
	`r4:&buffer+idx`
	`r5:x`

T	Data
	Return Address
	int buffer[Size]

Different Types of Information Flows



Different Types of Information Flows



Different levels for DIFT

- **Fine-grained** (processor level)

$$containers = addresses\ and\ registers$$

- **Medium-grained** (language level)

$$containers = variables$$

- **Coarse-grained** (operating system level)

$$containers = files, memory\ pages$$

OS-level Software DIFC (coarse-grained)

Description

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

Related Work

- Dedicated OS[1] : Asbestos, HiStar, Flume
- Modification of existing OS : **Blare**[2]

Pros & Cons

\+ Small runtime overhead ($< 10\%$)

\+ Kernel space isolation (hardware support) helps protecting the monitor

\- Overapproximation issue

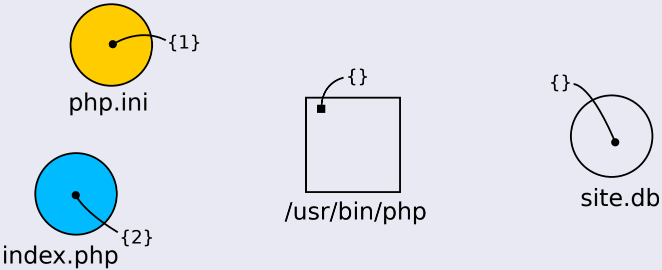
[1] Efstathopoulos et al. 2005; Zeldovich et al. 2006; Krohn et al. 2007.

[2] Geller et al. 2011; Hauser et al. 2012.

Blare: Tainting Information at the OS Level

Information tags

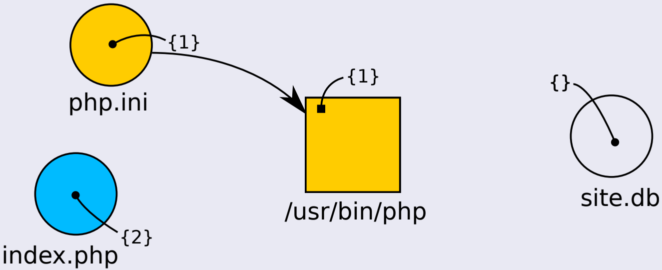
- Contain **meta-information**, describing content
- Updated after each information flow to describe the new content
- They trace the origin of the content



Blare: Tainting Information at the OS Level

Information tags

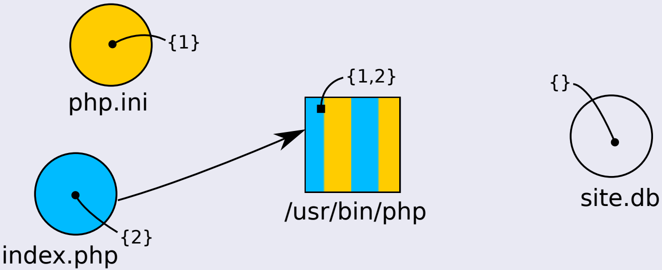
- Contain **meta-information**, describing content
- Updated after each information flow to describe the new content
- They trace the origin of the content



Blare: Tainting Information at the OS Level

Information tags

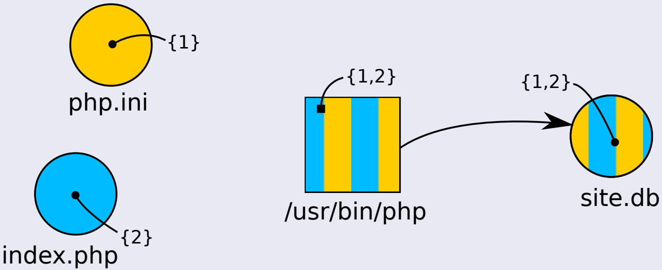
- Contain **meta-information**, describing content
- Updated after each information flow to describe the new content
- They trace the origin of the content



Blare: Tainting Information at the OS Level

Information tags

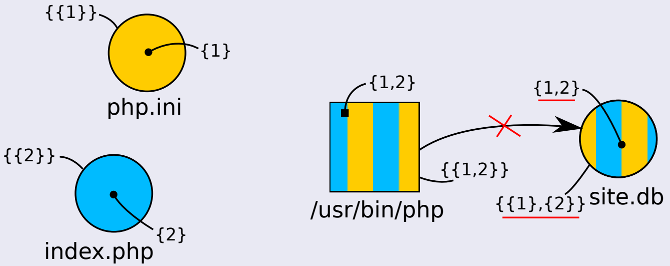
- Contain **meta-information**, describing content
- Updated after each information flow to describe the new content
- They trace the origin of the content



Blare: Controlling Information Flows at the OS Level

Policy tags

- Contain **meta-information**, describing the legal content of the containers
- It is set beforehand and doesn't change
- It is checked after each information flow and an alert is raised if the new content is not legal



Overapproximation Issue

KBlare considers processes as black boxes, so outputs are seen as a mix of all inputs. If we execute the code below:

```
f1 = open('file1', 'r')
f2 = open('file2', 'w')
f2.write(f1.read())

f3 = open('file3', 'r')
f4 = open('file4', 'w')
f4.write(f3.read())
```

then:

$$tags(f_2) = tags(f_1)$$
$$tags(f_4) = tags(f_1) \bigcup tags(f_3)$$

This is obviously an **overapproximation** due to the black box approach.

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

Description

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

Related Work

- Machine code[3]
- Specific language[4]

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

[3] Newsome and Song 2005; Harris, Jha, and Reps 2010.

[4] Chandra and Franz 2007; Nair et al. 2007.

Hardware-based DIFT (fine-grained)

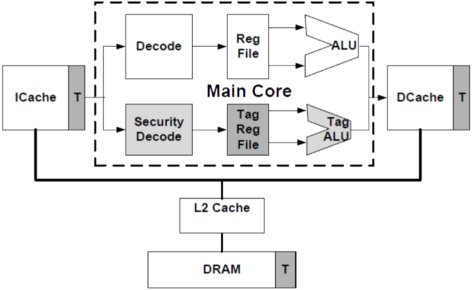


Figure: In-core DIFT [5]

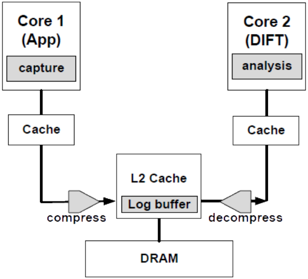


Figure: Dedicated CPU for DIFT [6]

[5] Dalton, Kannan, and Kozyrakis 2007.

[6] Nagarajan et al. 2008.

Hardware-based DIFT (fine-grained)

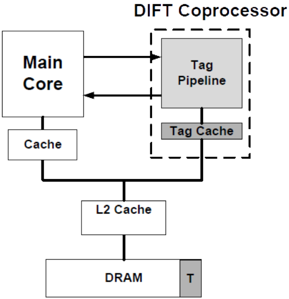


Figure: Dedicated DIFT co-processor [7]

[7] Kannan, Dalton, and Kozyrakis 2009.

Fine-grained DIFT : comparison of the existing approaches

		Advantages	Disadvantages
	Software	Flexible security policies	Runtime overhead (from 300% to 3700%)
HW-assisted	**In-core DIFT**	Low overhead (<10%)	Invasive modifications Few security policies
HW-assisted	**Dedicated CPU for DIFT**	Low overhead (<10%) Few modifications to CPU	Wasting resources Energy consumption (x 2)
HW-assisted	**Dedicated DIFT coprocessor**	Flexible security policies Low runtime overhead (<10%) CPU not modified	Communication between CPU and DIFT Coprocessor

HardBlare approach

Objectives

- Combine hardware level and OS level approaches
- Design and implement a realistic proof-of-concept
  - Unmodified (ASIC) main CPU (related work rely on softcores)
  - Dedicated DIFT coprocessor on FPGA
  - Rely on existing OS and applications (Linux system)

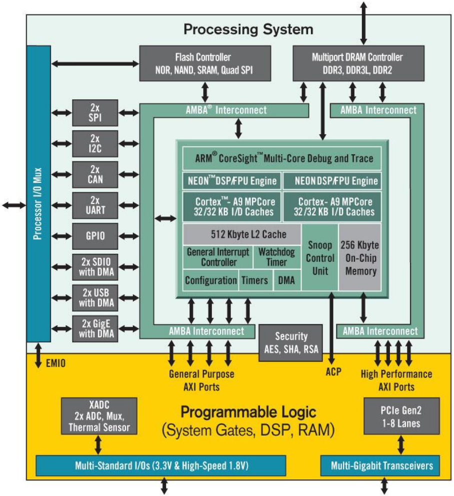
Technological choices

- Xilinx Zynq SoC (2 cores ARM Cortex A9 + FPGA)
- Dedicated Linux distribution using Yocto

Challenge

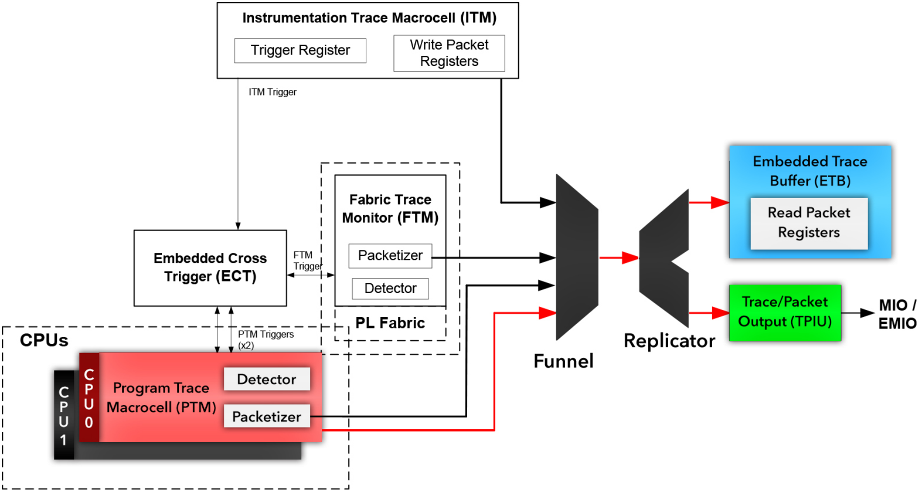
Semantic gap : limited visibility of CPU instructions on FPGA side

What can I do with my processor?

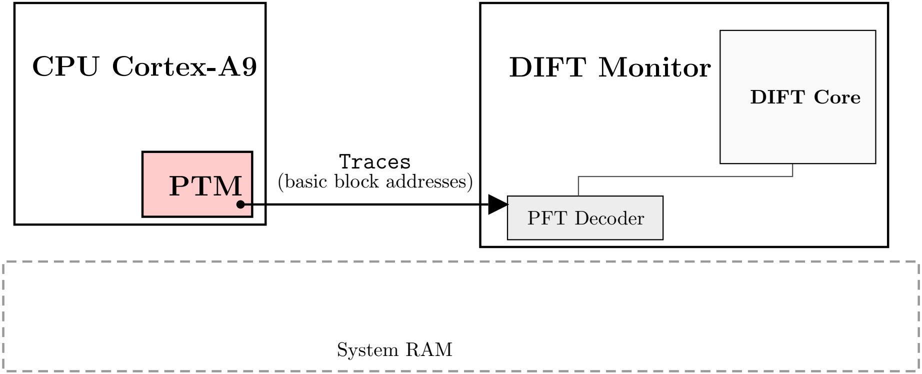


- CoreSight: debug components
- Available in most of Cortex-A + Cortex-M3 (for ARM)
- Can export debug-related infos

CoreSight components



PTM Traces

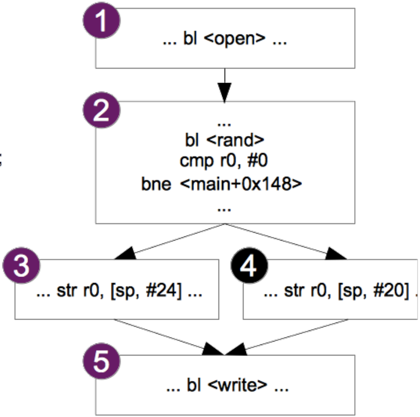

CPU Cortex-A9
PTM
Traces
(basic block addresses)
DIFT Monitor
DIFT Core
PFT Decoder
System RAM

PTM Traces

```
int main() {
  int file_public, file_secret, file_output;
  char public_buffer[1024];
  char secret_buffer[1024];
  char *temporary_buffer;
  file_public = open("files/public.txt",O_RDONLY);
  file_secret = open("files/secret.txt",O_RDONLY);
  file_output = open("files/output.txt",O_WRONLY);
  read(file_public, public_buffer, 1024);
  read(file_secret, secret_buffer, 1024);

  if( (rand() % 2) == 0){
      temporary_buffer = public_buffer;
  }
  else{
      temporary_buffer = secret_buffer;
  }

  write(file_output, temporary_buffer, 1024);
  return 0;
}
```



PTM trace : { 1 ; 2 ; 3 ; 5 }

Static Analysis

Problem

We need to know what's happened between two jumps

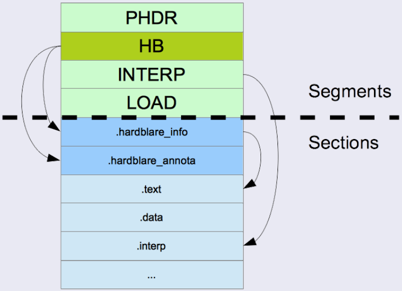
Solution

During compilation we also generate **annotations** that will be executed by the co-processor to propagate tags

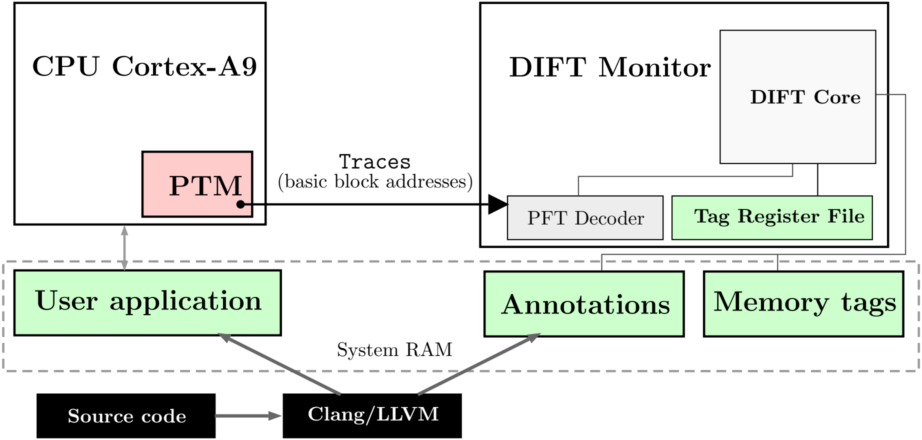
Examples :

add r0, r1, r2 $\Rightarrow$ $\underline{\texttt{r0}} \leftarrow \underline{\texttt{r1}} \cup \underline{\texttt{r2}}$

and r3, r4, r5 $\Rightarrow$ $\underline{\texttt{r3}} \leftarrow \underline{\texttt{r4}} \cup \underline{\texttt{r5}}$



Static Analysis



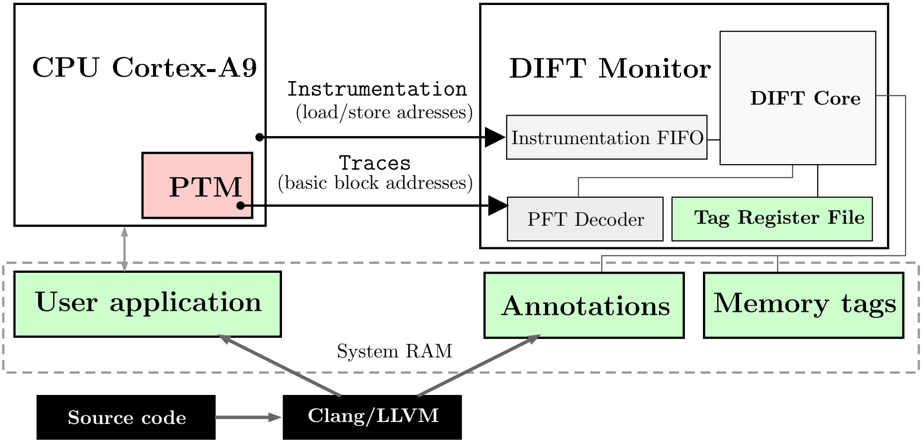
Instrumentation

- **Some addresses are resolved/calculated at run-time** :
  - *Solution :* **instrument the code**
- The instrumentation is done during the **last phase of the compilation** process.
- The register **r9 is dedicated** for the instrumentation.
- The instrumentation FIFO address is retrieved via a **UIO Driver**.

Examples :

```
ldr r0, [r2] ⇒  str r2, [r9]
                ldr r0, [r2]
str r3, [r4] ⇒  str r5, [r9]
                str r3, [r5]
```

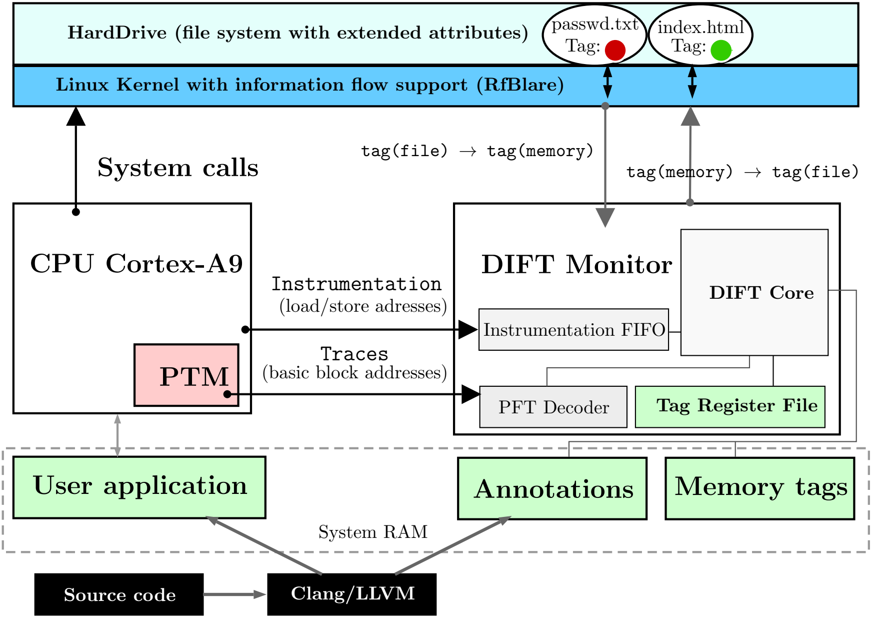
Instrumentation



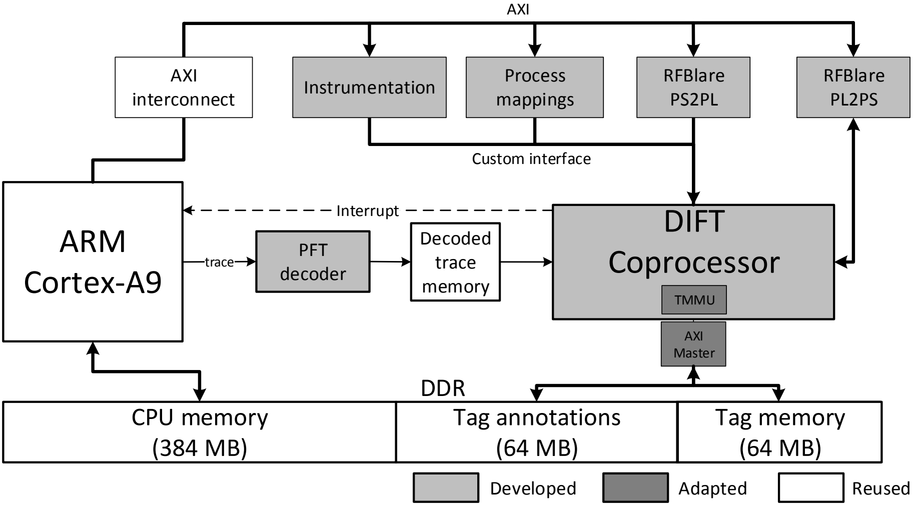
RfBlare: System calls

- **Problem:** We want to transmit tags from/to the operating system.
  Solution: **Linux Security Modules Hooks**
- **Problem:** We want to persistently store tags in the system.
  Solution: **Extended file attributes**
- When **reading** data from a file.
  We are propagating the tag of **the read file to the destination buffer**.
- When **writing** data to a file.
  We are propagating the tag of **the source buffer to the destination file**.

RfBlare: System calls



Overall architecture of the DIFT monitor



What does a trace looks like?

Code Source

```
int i;
for (i = 0; i < 10; i++)
```

What does a trace looks like?

Code Source

```
int i;
for (i = 0; i < 10; i++)
```

Assembly

8638 `for_loop:`

`...`

`b` 8654:

`...`

866c: `bcc` 8654

What does a trace looks like?

Code Source

```
int i;
for (i = 0; i < 10; i++)
```

Assembly

8638 `for_loop:`

...

`b` 8654:

...

866c: `bcc` 8654

Trace

00 00 00 00 00 80 08 38 86 00 00
21 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a
86 01 00 00 00 00 00 00 00 00

What does a trace looks like?

Code Source

```
int i;
for (i = 0; i < 10; i++)
```

Assembly

8638 `for_loop:`

`...`

`b` 8654:

`...`

866c: `bcc` 8654

Trace

00 00 00 00 00 80 08 38 86 00 00
21 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a
86 01 00 00 00 00 00 00 00 00

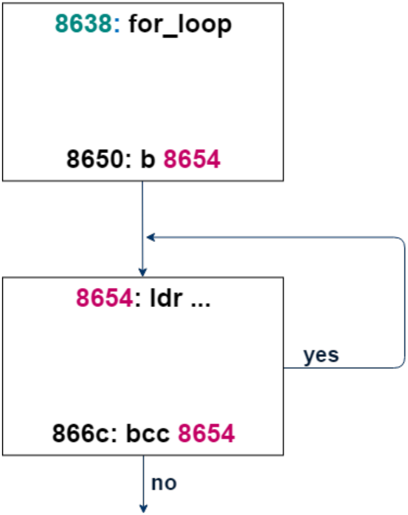
Decoded Trace

A-sync

Address 00008638, (I-sync Context 00000000, IB 21)

Address 00008654, Branch Address packet (x 10)

What does a trace looks like?



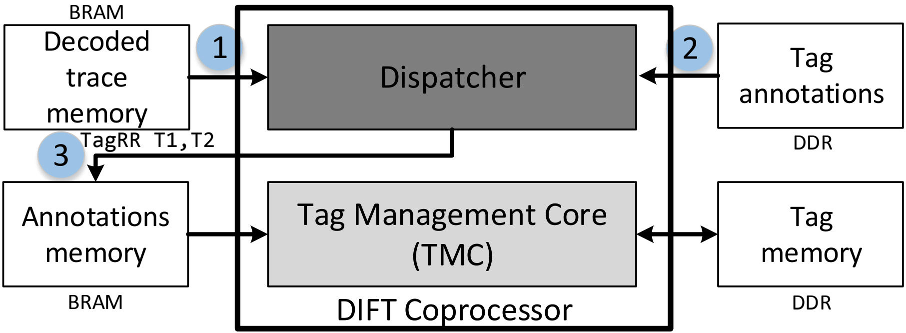
Decoded Trace

A-sync

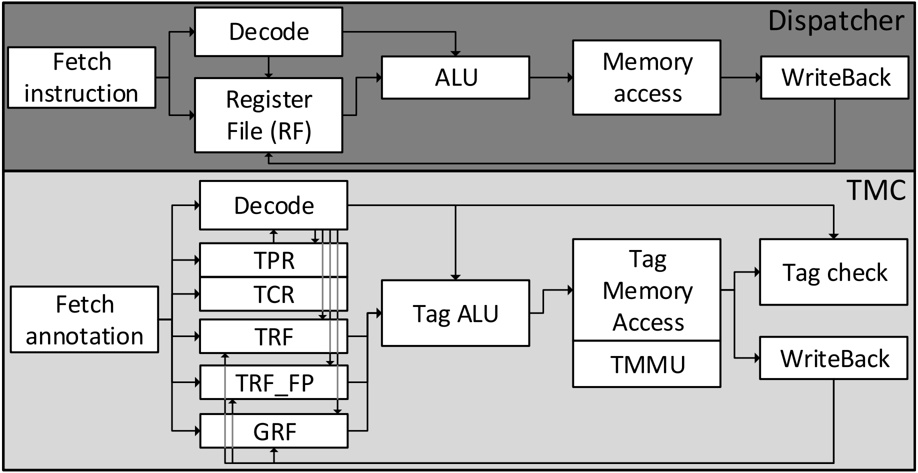
Address 00008638, (I-sync Context 00000000, IB 21)

Address 00008654, Branch Address packet (x 10)

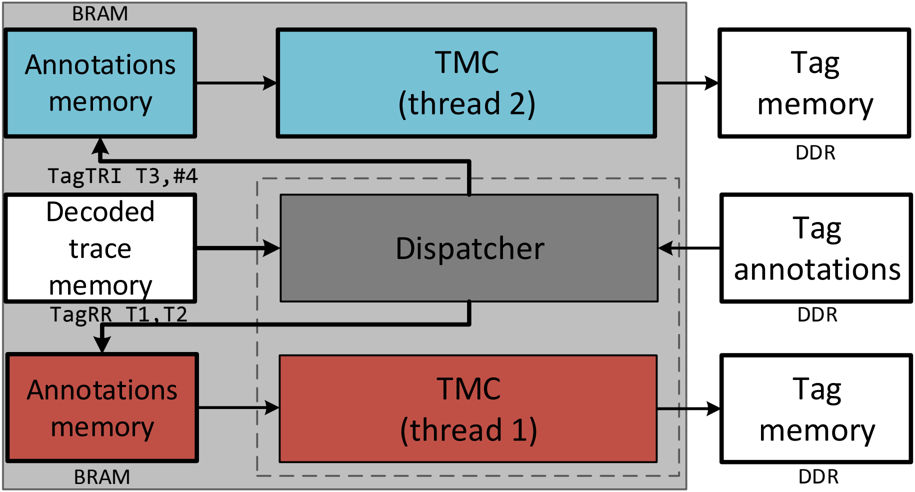
Dedicated DIFT coprocessor



Internal structure of the DIFT coprocessor



Extension for 2 threads - DIFT coprocessor



Extension for 2 threads - Trace details

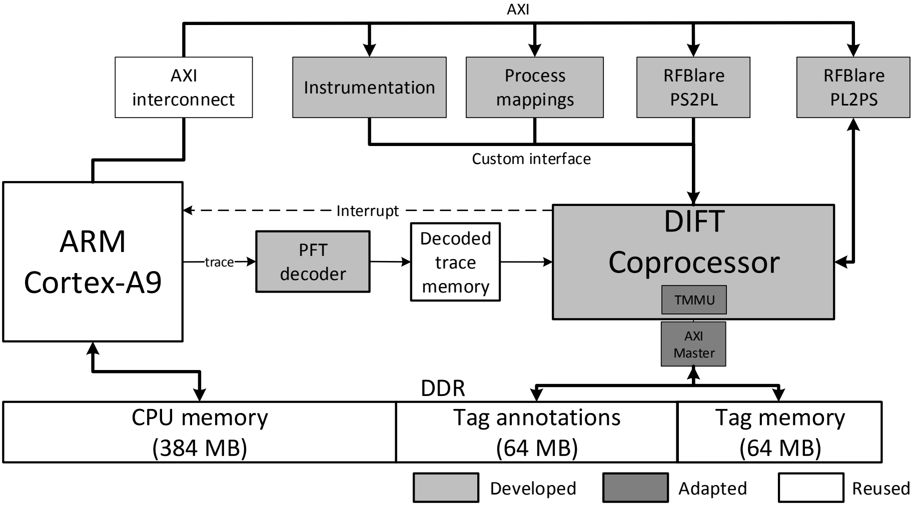
Trace

00 00 00 00 00 *80 08 74 05 01 00 21* ***<u>42 d2 04 00</u>*** 95 04 *08 84 05 01 00 21* ***<u>42 d2 04 00</u>*** e5 03 *08 98 05 01 00 21* ***<u>42 d2 04 00</u>*** fd 03 *08 74 05 01 00 21* ***<u>42 d3 04 00</u>*** 95 04 *08 84 05 01 00 21* ***<u>42 d3 04 00</u>***

A-sync *I-sync* Branch address packet

Decoded trace	**Context ID**	**Stored address**
00010574	0004d2 42	00010574
00010428	0004d2 42	00010428
00010584	0004d2 42	00010584
000103c8	0004d2 42	000103c8
00010598	0004d2 42	00010598
000103f8	0004d2 42	000103f8
00010574	0004d3 42	0001057**<u>5</u>**
00010428	0004d3 42	0001042**<u>9</u>**
00010584	0004d3 42	0001058**<u>5</u>**

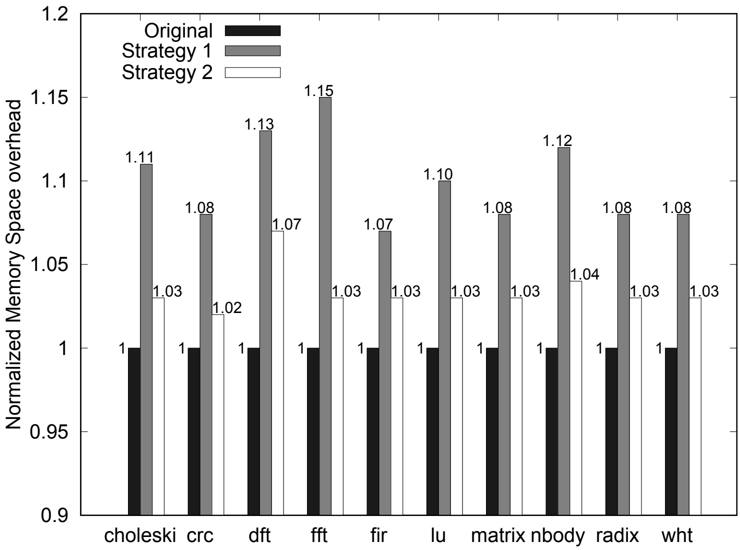
Overall architecture of the DIFT monitor



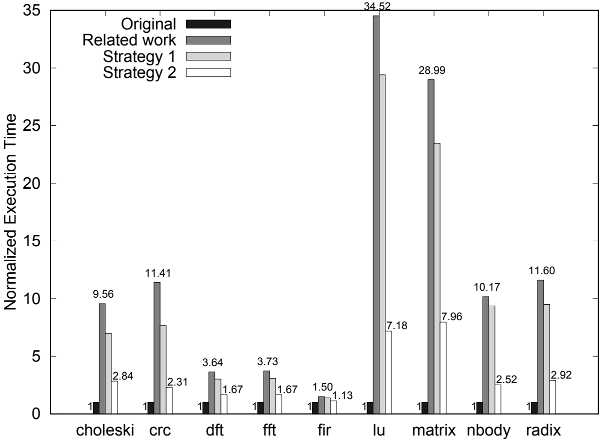
Area results (single-thread implementation)

IP Name	**Slice LUTs (in %)**	**Slice Registers (in %)**	**BRAM Tile**
Dispatcher	2223 (4.18%)	1867 (1.75%)	3
TMC	1837 (3.45%)	2581 (2.43%)	6
PFT Decoder	121 (0.23%)	231 (0.22%)	0
Instrumentation	676 (1.27%)	2108 (1.98%)	0
Blare PS2PL	662 (1.24%)	2106 (1.98%)	0
Blare PL2PS	62 (0.12%)	56 (0.05 %)	0
Decoded trace memory	0	0	2
AXI Master	858 (1.61%)	2223 (2.09 %)	0
TMMU	295 (0.55%)	112(0.10 %)	3
AXI Interconnect	2733 (5.14%)	2495 (2.34 %)	0
Miscellaneous	1381 (2.6%)	2160 (2.03%)	0
Total Design	**10848 (20.39%)**	**15939 (14.98%)**	**14 (10%)**
Total Available	**53200**	**106400**	**140**

Memory footprint



Some latency results



Quick note

Enabling CoreSight components ⇒ Nearly no cost in terms of latency.
Latency is only due to DIFT-related computations.

Comparison with existing works

Approaches	Kannan [8]	Deng [9]	Heo [10]	Wahab [11]	Latest work
Hardcore portability	No	No	Yes	Yes	Yes
Main CPU	Softcore	Softcore	Softcore	Hardcore	Hardcore
Communication overhead	N/A	N/A	60%	5.4%	335%
Library instrumentation	N/A	N/A	partial	No	Yes
All information flows	Yes	Yes	No	No	Yes
Area overhead	6.4%	14.8%	14.47%	0.47%	0.95 %
Power overhead	N/A	6.3%	24%	16%	16.2%
Max frequency	N/A	256 MHz	N/A	250 MHz	250 MHz
FP support	No	No	No	No	Yes
Multi-threaded support	No	No	No	No	Yes

[8] Kannan, Dalton, and Kozyrakis 2009.
[9] Deng and Suh 2012.
[10] Heo et al. 2015.
[11] Wahab et al. 2017.

Perspectives

Take away:

- CoreSight PTM allows to obtain runtime information (Program Flow)
- Non-intrusive tracing $\Rightarrow$ Negligible performance overhead
- Support for multi-threaded and floating-point software
- Kernel support with RfBlare

Perspectives:

- Full PoC later this year (SoC files + Yocto)
- Intel / ST? (study)
- Multicore multi-thread IFT

HardBlare, a hardware/software co-design approach for Information Flow Control

Guillaume Hiet [α], Pascal Cotret [β]

[α] CentraleSupélec, guillaume.hiet@centralesupelec.fr
[β] R&D engineer, pascal.cotret@gmail.com

June 22, 2018

Many thanks to Muhammad, Mounir, Arnab, Vianney and Guy :)

https://hardblare.cominlabs.u-bretagneloire.fr

Bibliography I

Dalton, Kannan, and Kozyrakis 2007. Raksha: A Flexible Information Flow Architecture for Software Security.

Nagarajan et al. 2008. Dynamic Information Tracking on Multicores.

Kannan, Dalton, and Kozyrakis 2009. Decoupling dynamic information flow tracking with a dedicated coprocessor.

Deng and Suh 2012. High-performance parallel accelerator for flexible and efficient run-time monitoring.

Heo et al. 2015. Implementing an Application-Specific Instruction-Set Processor for System-Level Dynamic Program Analysis Engines.

Wahab et al. 2017. ARMHEx: A hardware extension for DIFT on ARM-based SoCs.

Bibliography II

Efstathopoulos et al. 2005. Labels and event processes in the asbestos operating system.

Zeldovich et al. 2006. Making information flow explicit in HiStar.

Krohn et al. 2007. Information Flow Control for Standard OS Abstractions.

Geller et al. 2011. Information Flow Control for Intrusion Detection derived from MAC policy.

Hauser et al. 2012. A taint marking approach to confidentiality violation detection.

Newsome and Song 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software.

Bibliography III

Harris, Jha, and Reps 2010. DIFC programs by automatic instrumentation.

Chandra and Franz 2007. Fine-Grained Information Flow Analysis and Enforcement in a Java Virtual Machine.

Nair et al. 2007. A Virtual Machine Based Information Flow Control System for Policy Enforcement.

Backup slides

TrustZone support