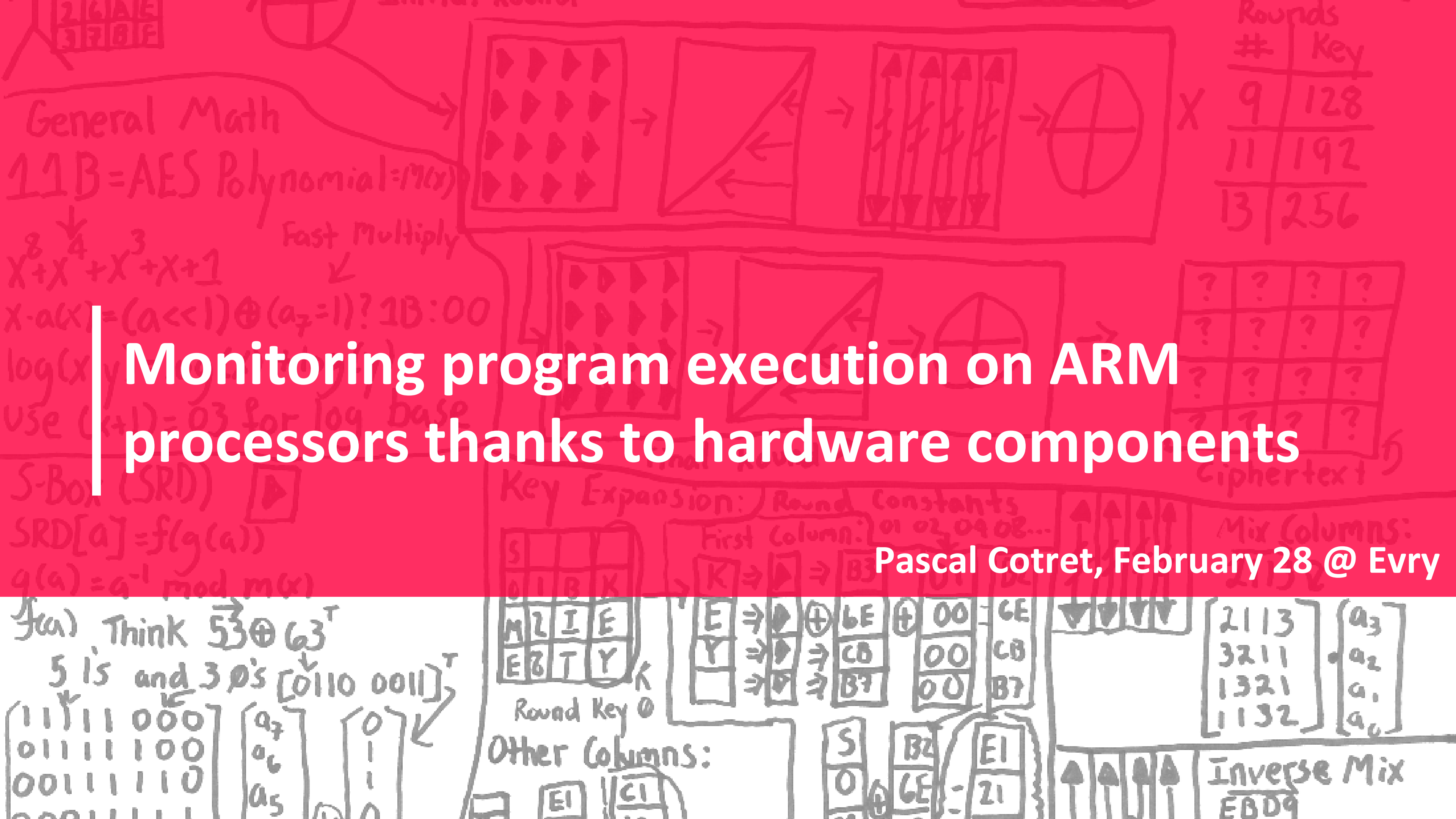


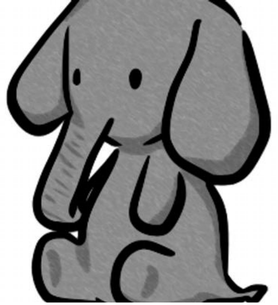
Monitoring program execution on ARM processors thanks to hardware components

Pascal Cotret, February 28 @ Evry

Hello!

I am Pascal Cotret

- Embedded software security engineer
- Research in my spare-time



HardBlare (2015/2018)

Funded by Labex CominLabs

- 3 labs (INRIA CIDRE, Lab-STICC, SCEE/IETR)
- 2 PhD students.
- 1 postdoc (from February 2018).

=> Heterogeneous information flow control

Dynamic Information Flow Tracking

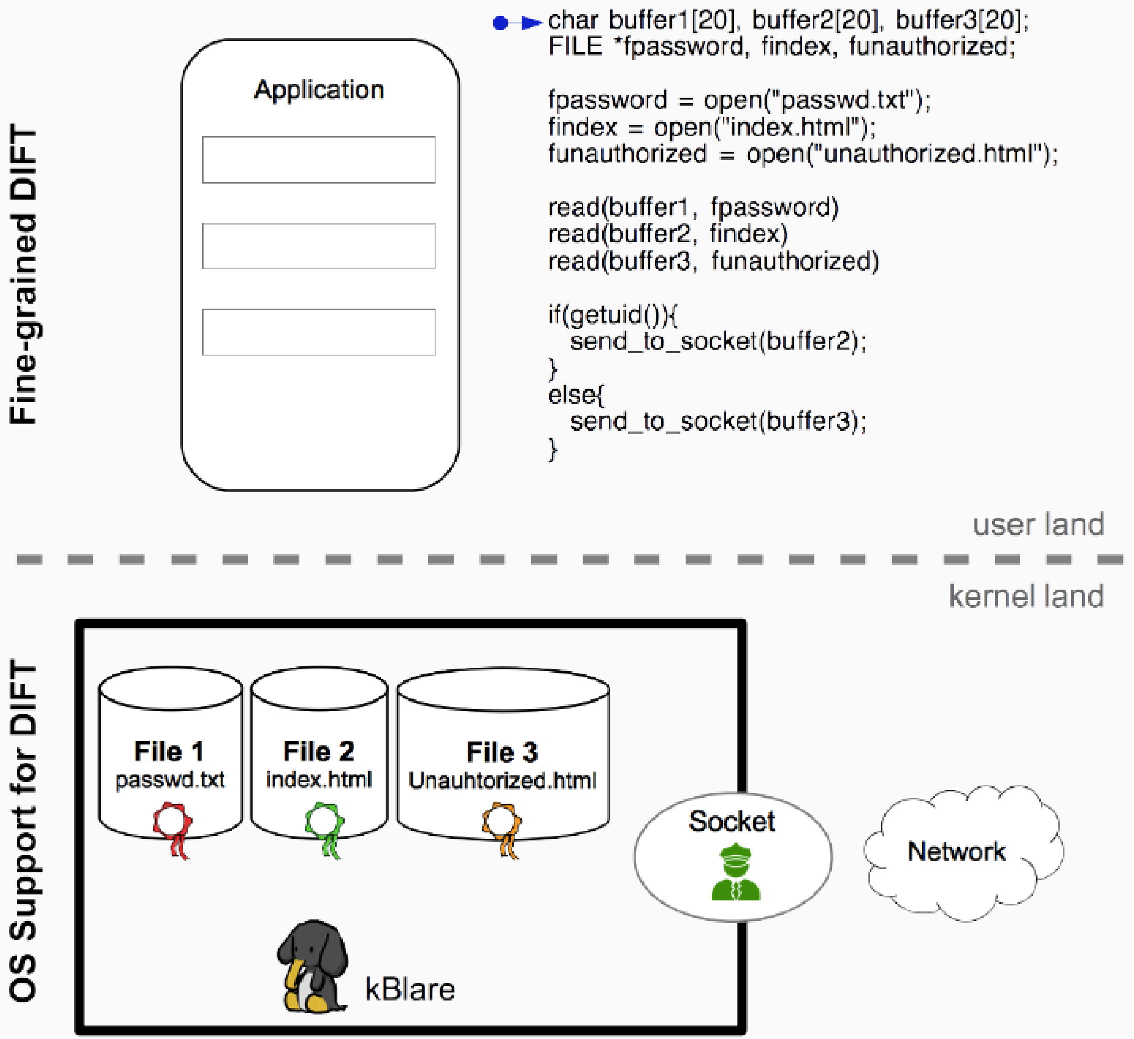
Dynamic Information Flow Tracking (DIFT) is a promising technique for detecting software attacks.

Motivation: DIFT for security purposes => Integrity and Confidentiality

DIFT principle:

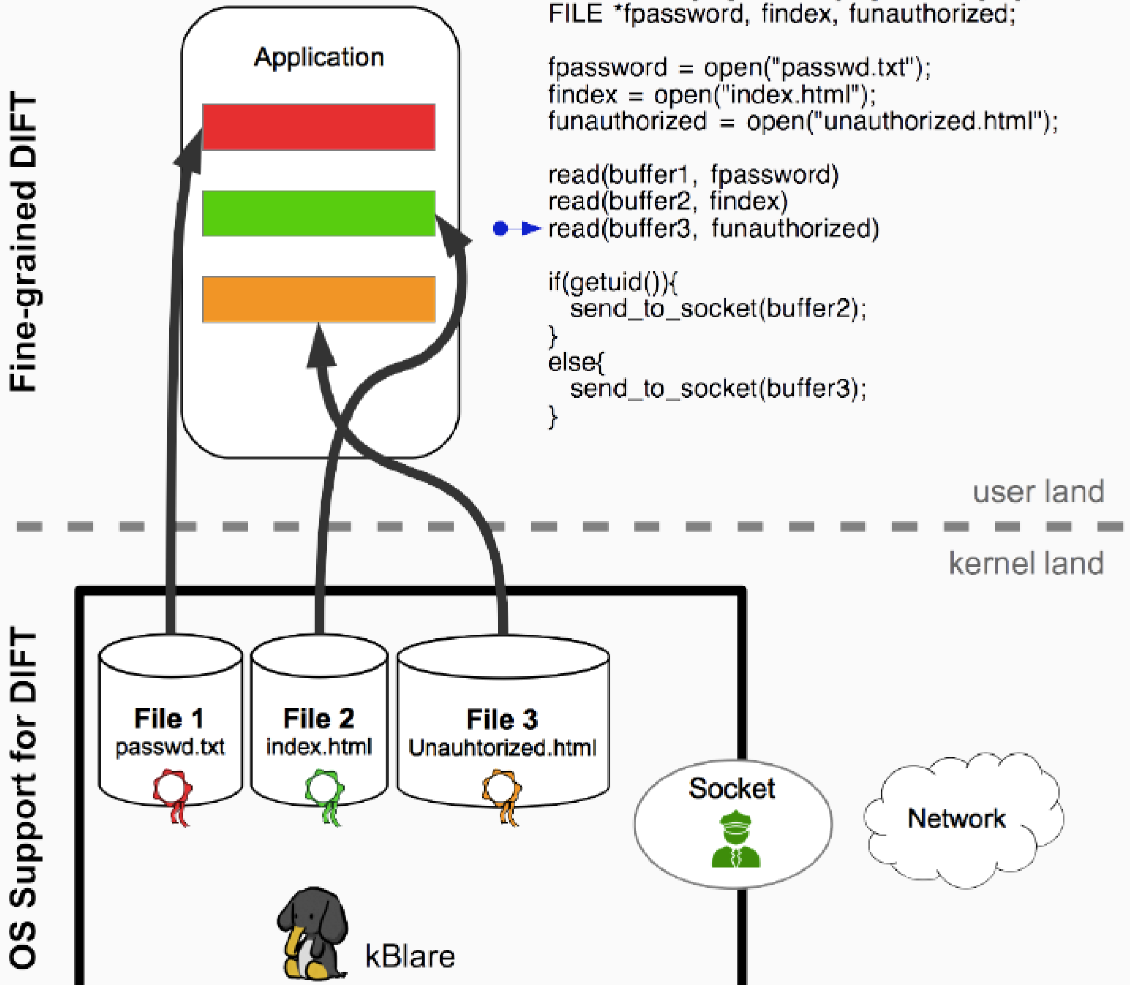
- We attach tags to containers and specify an information flow policy, i.e. relations between tags.
- At runtime, we propagate tags to reflect information flows that occurs.
- Allows to detect any security policy violation at run-time.

DIFT



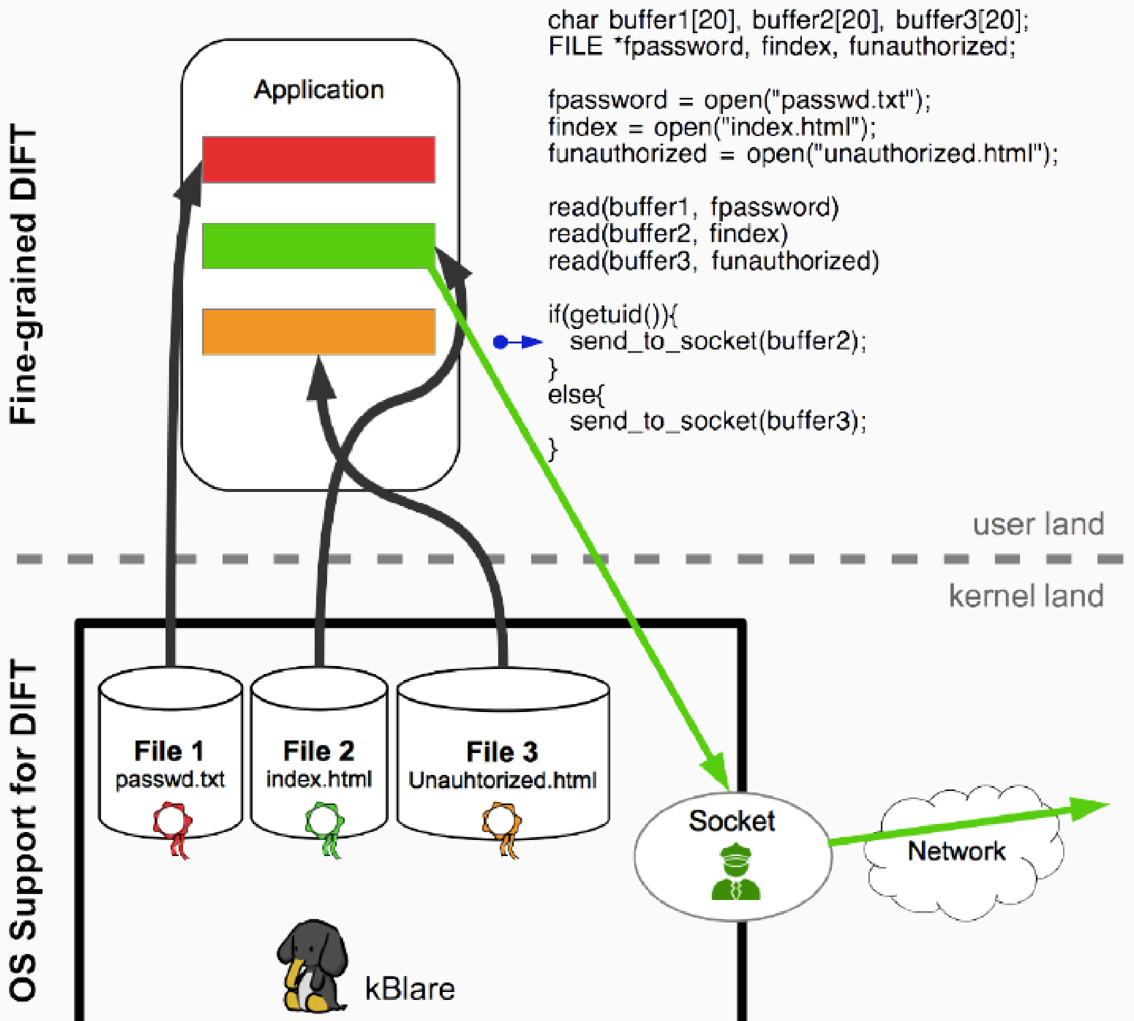
```
char buffer1[20], buffer2[20], buffer3[20];
FILE *fpassword, findex, funauthorized;

fpassword = open("passwd.txt");
findex = open("index.html");
funauthorized = open("unauthorized.html");

read(buffer1, fpassword)
read(buffer2, findex)
read(buffer3, funauthorized)

if(getuid()){
  send_to_socket(buffer2);
}
else{
  send_to_socket(buffer3);
}
```

DIFT



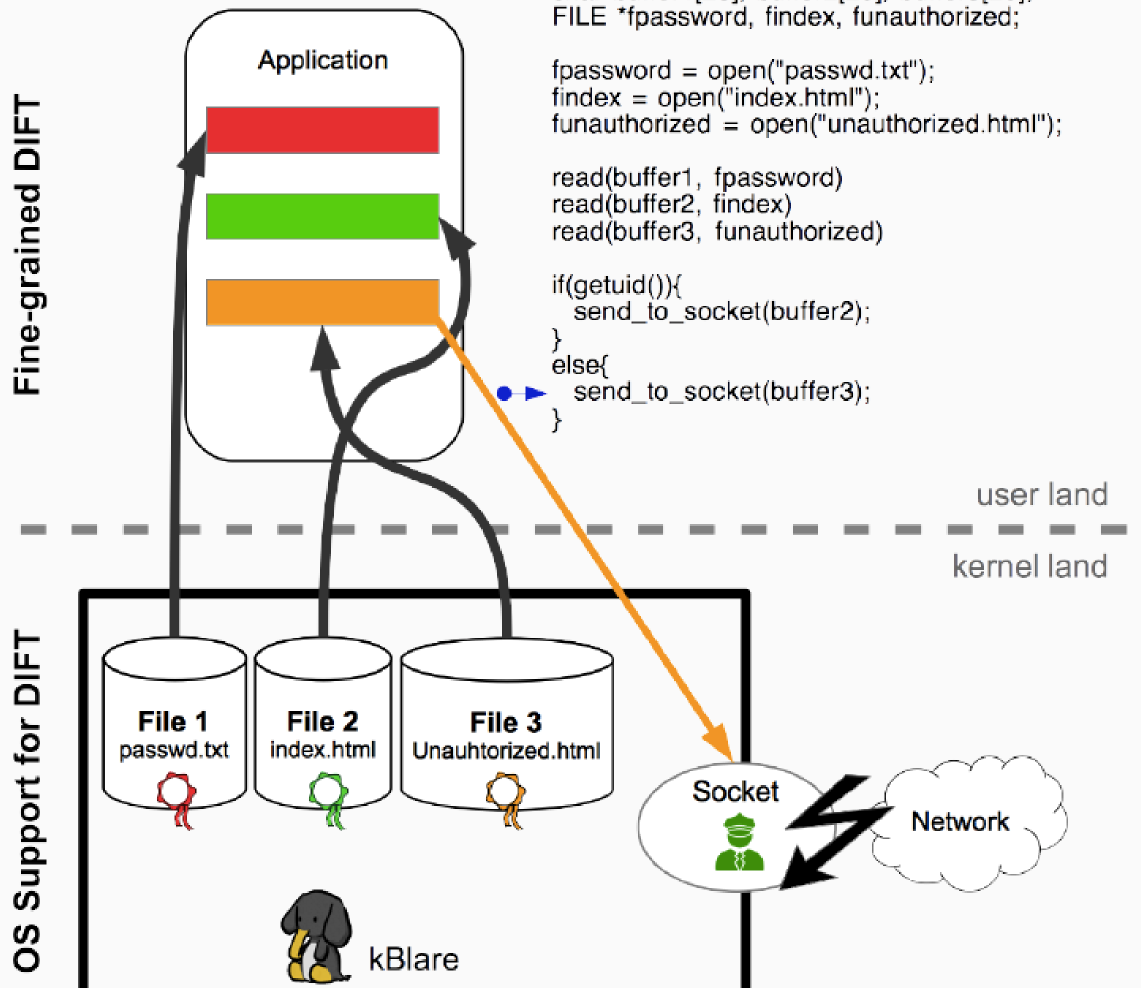
```
char buffer1[20], buffer2[20], buffer3[20];
FILE *fpassword, findex, funauthorized;

fpassword = open("passwd.txt");
findex = open("index.html");
funauthorized = open("unauthorized.html");

read(buffer1, fpassword)
read(buffer2, findex)
read(buffer3, funauthorized)

if(getuid()){
  send_to_socket(buffer2);
}
else{
  send_to_socket(buffer3);
}
```

DIFT



```
char buffer1[20], buffer2[20], buffer3[20];
FILE *fpassword, findex, funauthorized;

fpassword = open("passwd.txt");
findex = open("index.html");
funauthorized = open("unauthorized.html");

read(buffer1, fpassword)
read(buffer2, findex)
read(buffer3, funauthorized)

if(getuid()){
  send_to_socket(buffer2);
}
else{
  send_to_socket(buffer3);
}
```

DIFT



```
char buffer1[20], buffer2[20], buffer3[20];
FILE *fpassword, findex, funauthorized;

fpassword = open("passwd.txt");
findex = open("index.html");
funauthorized = open("unauthorized.html");

read(buffer1, fpassword)
read(buffer2, findex)
read(buffer3, funauthorized)

if(getuid()){
  send_to_socket(buffer2);
}
else{
  send_to_socket(buffer3);
}
```

DIFT – Three main steps

- **Tag initialization:** data are tagged with theirs "security level"

```
password="abcd"  Tag(password)=secret
```

- **Tag propagation:** any new data derived from the tagged data is also tagged

```
log=err+password Tag(log)=Tag(err)+Tag(password)
```

- **Tag check:** raise an exception if an information flow doesn't respect a security policy

```
write(log,network) Policy: (Tag(log)==public)
```

Different levels for DIFT

- Fine-grained (processor level)

`vi = addresses and registers; P = instructions`

- Medium-grained (language level)

`vi = variables; P = functions`

- Coarse-grained (operating system level)

`vi = files; P = executables`

DIFT

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

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

T	Data
🟩	Return Address
🟩	int buffer[Size]

DIFT

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

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

T	Data
🟩	Return Address
🟩	int buffer[Size]

DIFT

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

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

T	Data
🟩	Return Address
🟩	int buffer[Size]

DIFT

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

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

T	Data
🟩	Return Address
🟩	int buffer[Size]

DIFT

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

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

T	Data
🟥	Return Address
🟩	int buffer[Size]

Related work

Different levels:

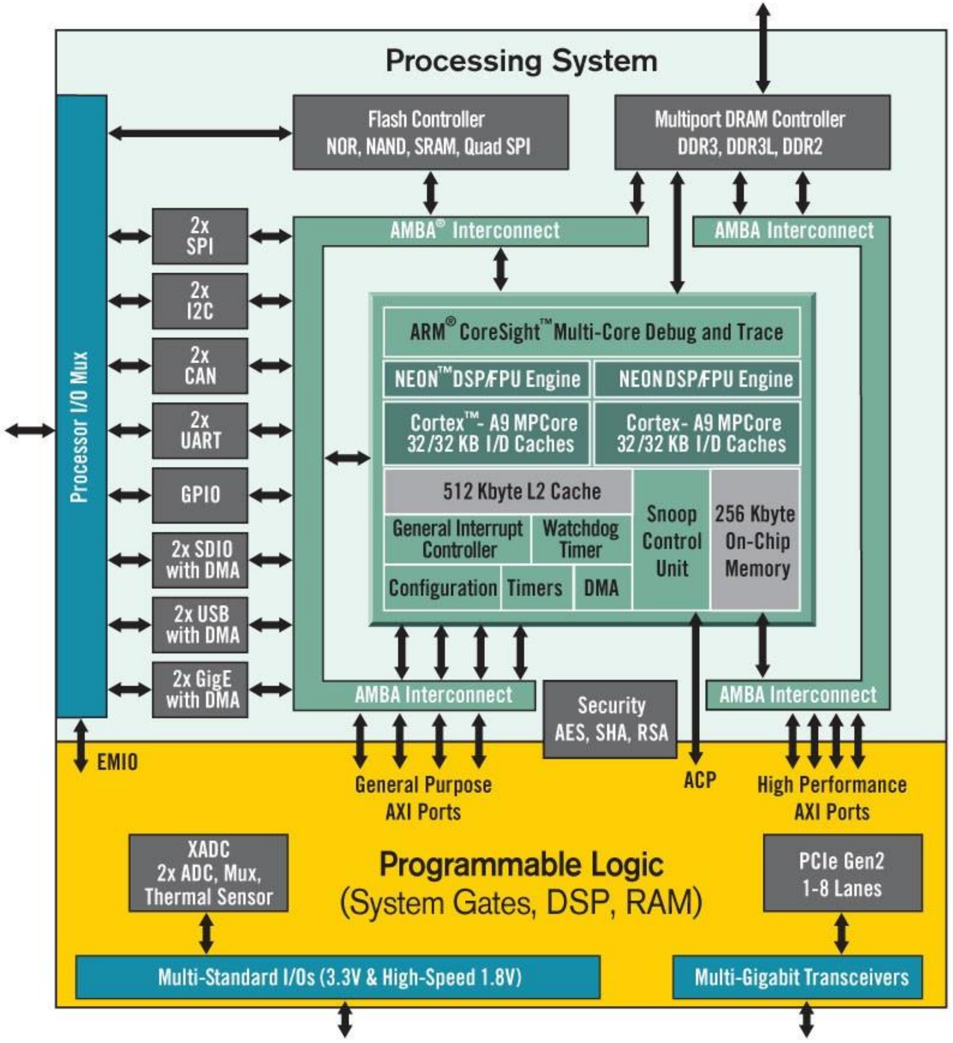
- Application level
  - Java / Android, Javascript, C
- OS level
  - Laminar, HiStar
  - kBlare (1)
- Low level
  - Raksha (Kannan et al.)
  - Flexitaint (Ventakaramani et al.)
  - Flexcore (Deng et al.)
  - PAU (Heo et al.)

(1) Jacob Zimmermann, Ludovic Mé, and Christophe Bidan. Introducing Reference Flow Control for Detecting Intrusion Symptoms at the OS Level. In : RAID 2002.

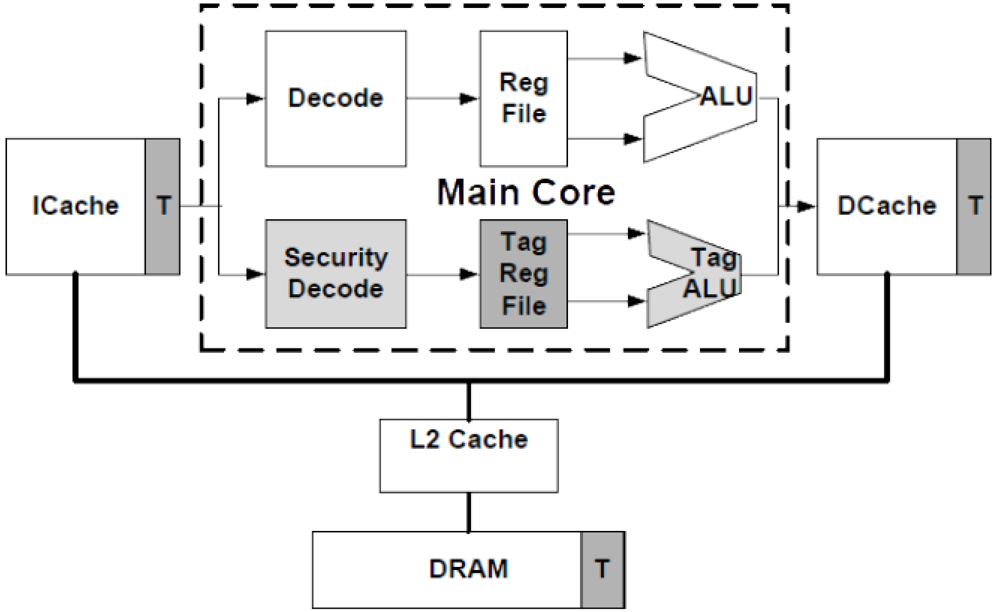
Target architecture

SoC = System-on-Chip

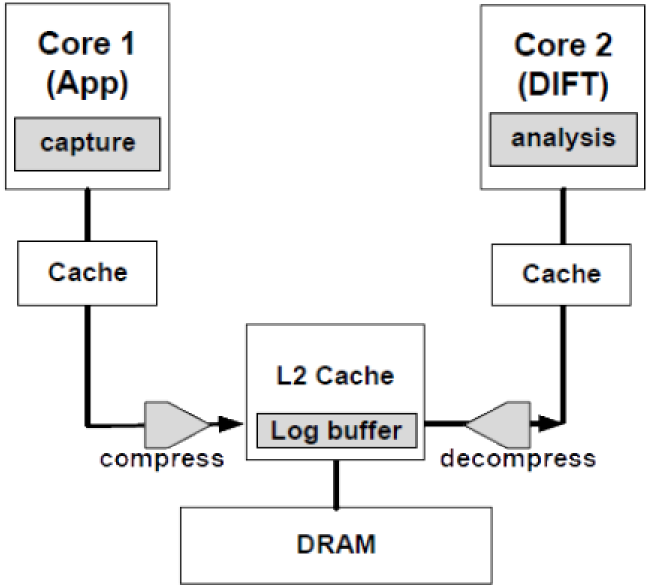
- 1+ processor(s)
- Configurable electronic chip (aka FPGA)


Processing System
Flash Controller NOR, NAND, SRAM, Quad SPI
Multiport DRAM Controller DDR3, DDR3L, DDR2
AMBA® Interconnect
AMBA Interconnect
2x SPI
2x I2C
2x CAN
2x UART
GPIO
2x SDIO with DMA
2x USB with DMA
2x GigE with DMA
Processor I/O Mux
ARM® CoreSight™ Multi-Core Debug and Trace
NEON™ DSP/FPU Engine
NEON DSP/FPU Engine
Cortex™- A9 MPCore 32/32 KB I/D Caches
Cortex- A9 MPCore 32/32 KB I/D Caches
512 Kbyte L2 Cache
General Interrupt Controller
Watchdog Timer
Configuration
Timers
DMA
Snoop Control Unit
256 Kbyte On-Chip Memory
AMBA Interconnect
Security AES, SHA, RSA
AMBA Interconnect
EMIO
General Purpose AXI Ports
ACP
High Performance AXI Ports
XADC 2x ADC, Mux, Thermal Sensor
Programmable Logic (System Gates, DSP, RAM)
PCIe Gen2 1-8 Lanes
Multi-Standard I/Os (3.3V & High-Speed 1.8V)
Multi-Gigabit Transceivers

Related work

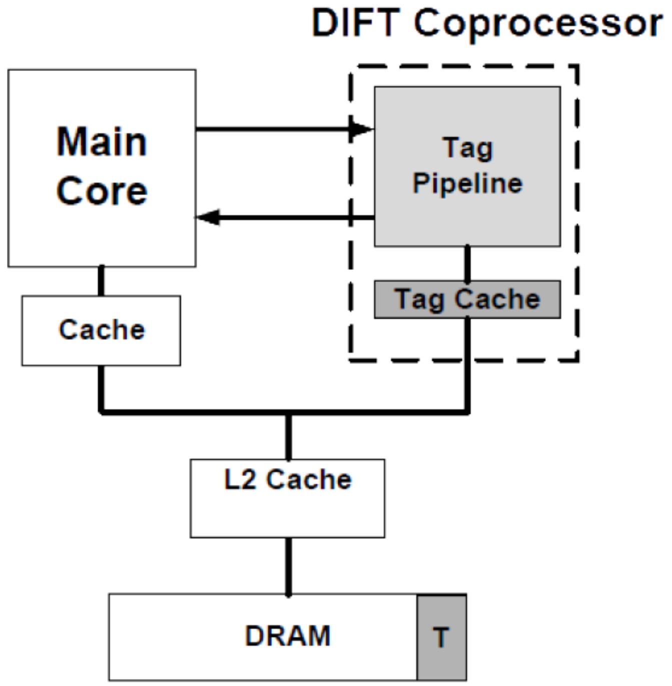


In-core DIFT



Offloading DIFT

Related work



Off-core DIFT

(2) Hari Kannan, Michael Dalton, and Christos ozyrakis. Decoupling dynamic information flow tracking with a dedicated coprocessor. In : Dependable Systems & Networks, 2009. IEEE. 2009, pp. 105-114.

Related work

	Advantages	Disadvantages
Software	Flexible security policies Multiple attacks detected	Overhead (from 300% to 3700%)
In-core DIFT	Low overhead (<10%)	Invasive modifications Few security policies
Dedicated CPU for DIFT	Low overhead (<10%) Few modifications to CPU	Wasting resources Energy consumption (x2)
Dedicated DIFT coprocessor	Flexible security policies Low overhead (<10%) CPU not modified	CPU/coprocessor communication

Related work

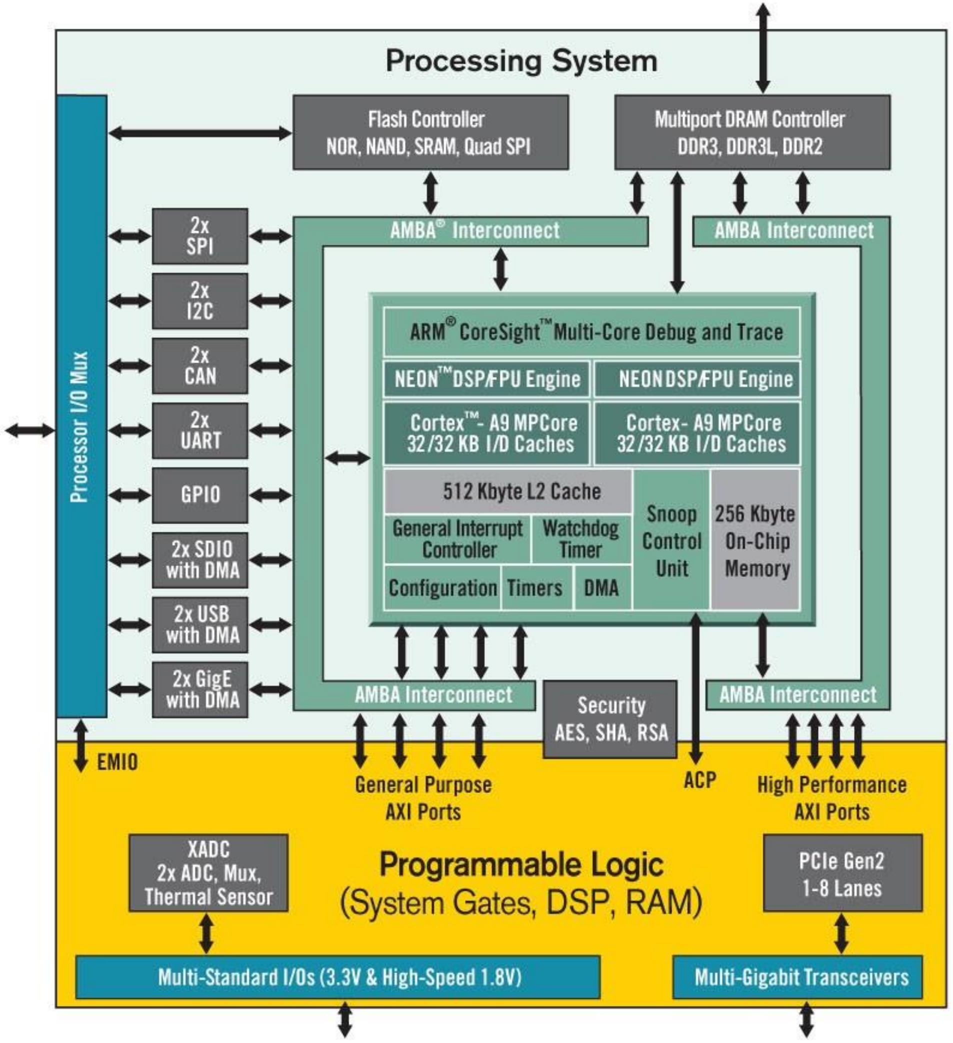
ARMHEx approach:

- Reduce overhead of software instrumentation as it represents the major portion of overall DIFT execution time overhead
- Lack of security of DIFT coprocessor
- No existing work targets ARM-based SoCs
  (related work implementations on softcores)
- Additional challenges
  - Limited visibility
  - Frequency gap between CPU and DIFT coprocessor
  - Communication interface…

Target architecture

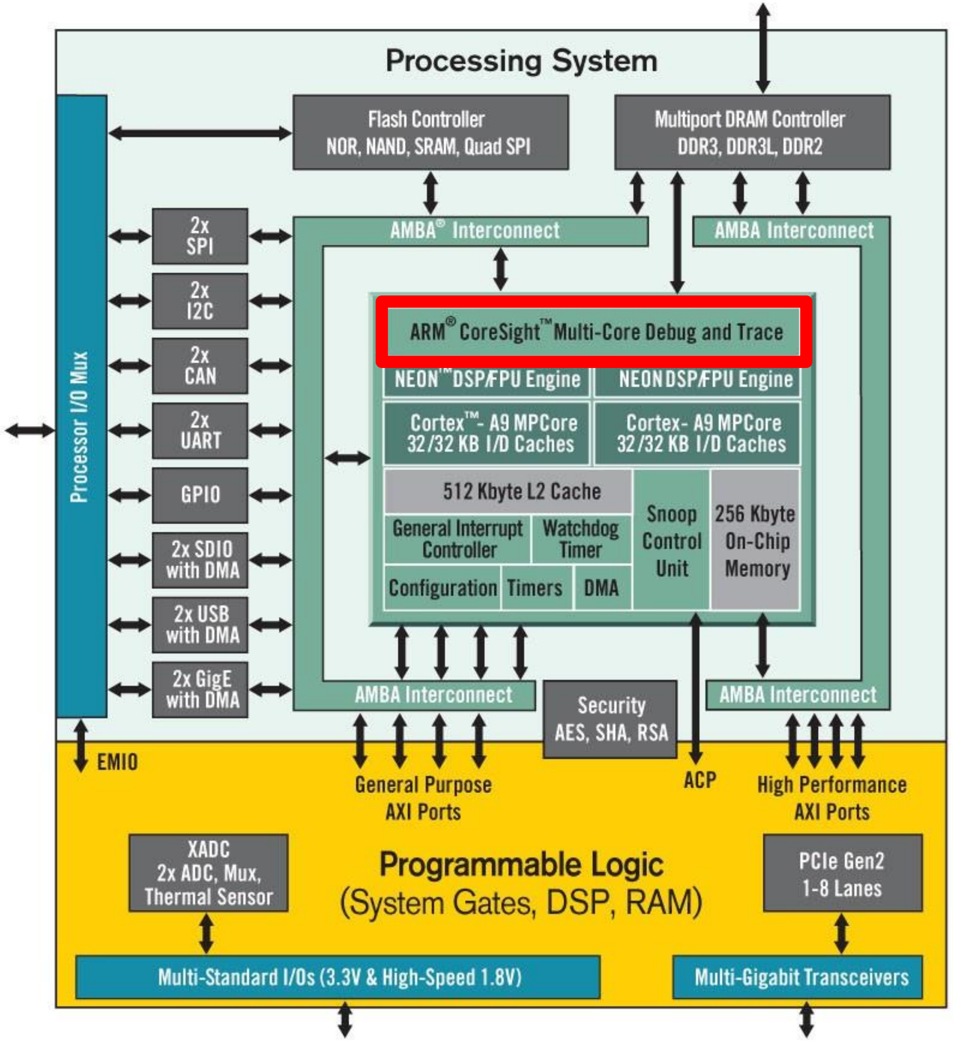
SoC = System-on-Chip

- 1+ processor(s)
- Configurable electronic chip (aka FPGA)

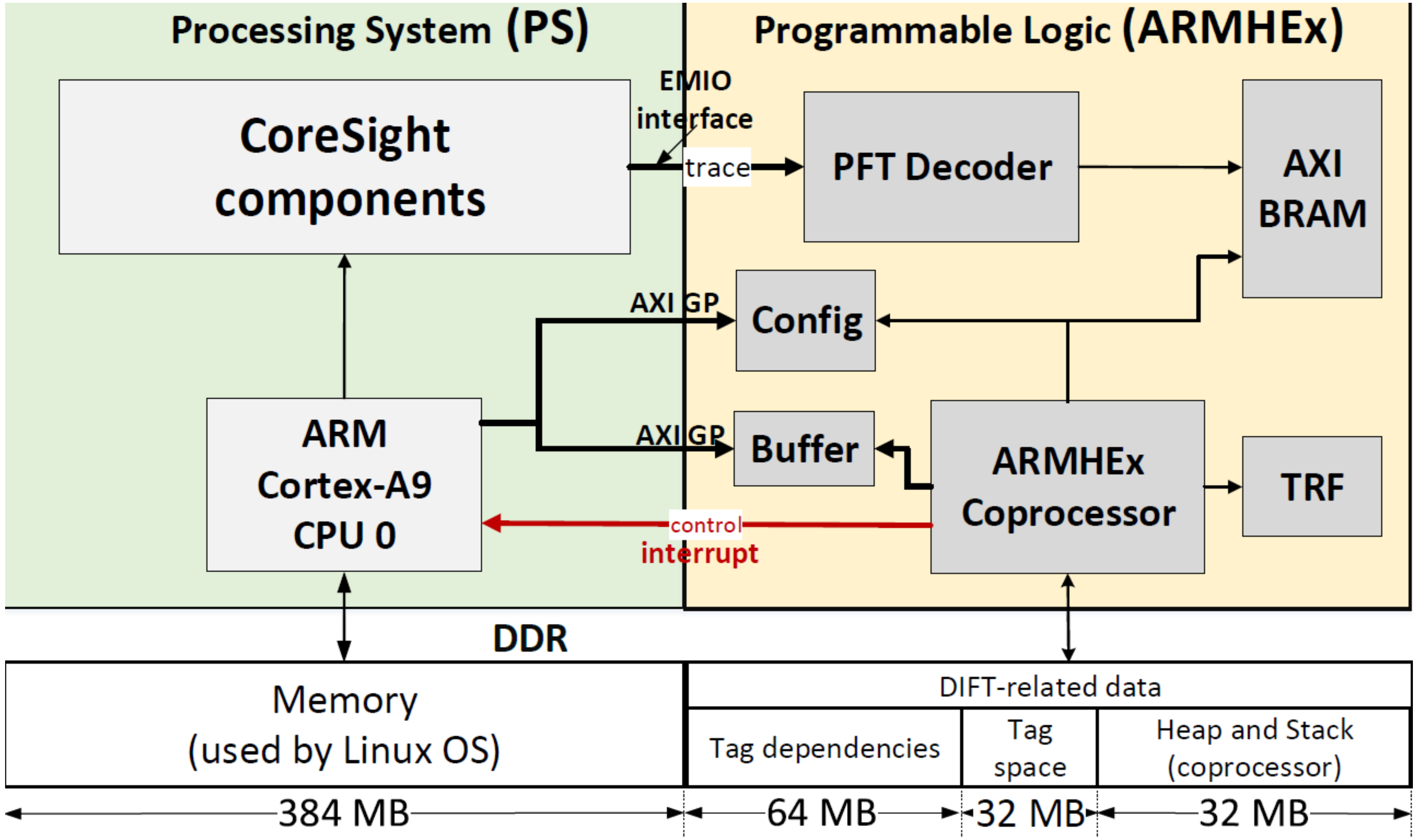


CoreSight components

A set of IP blocks providing HW-assisted system tracing

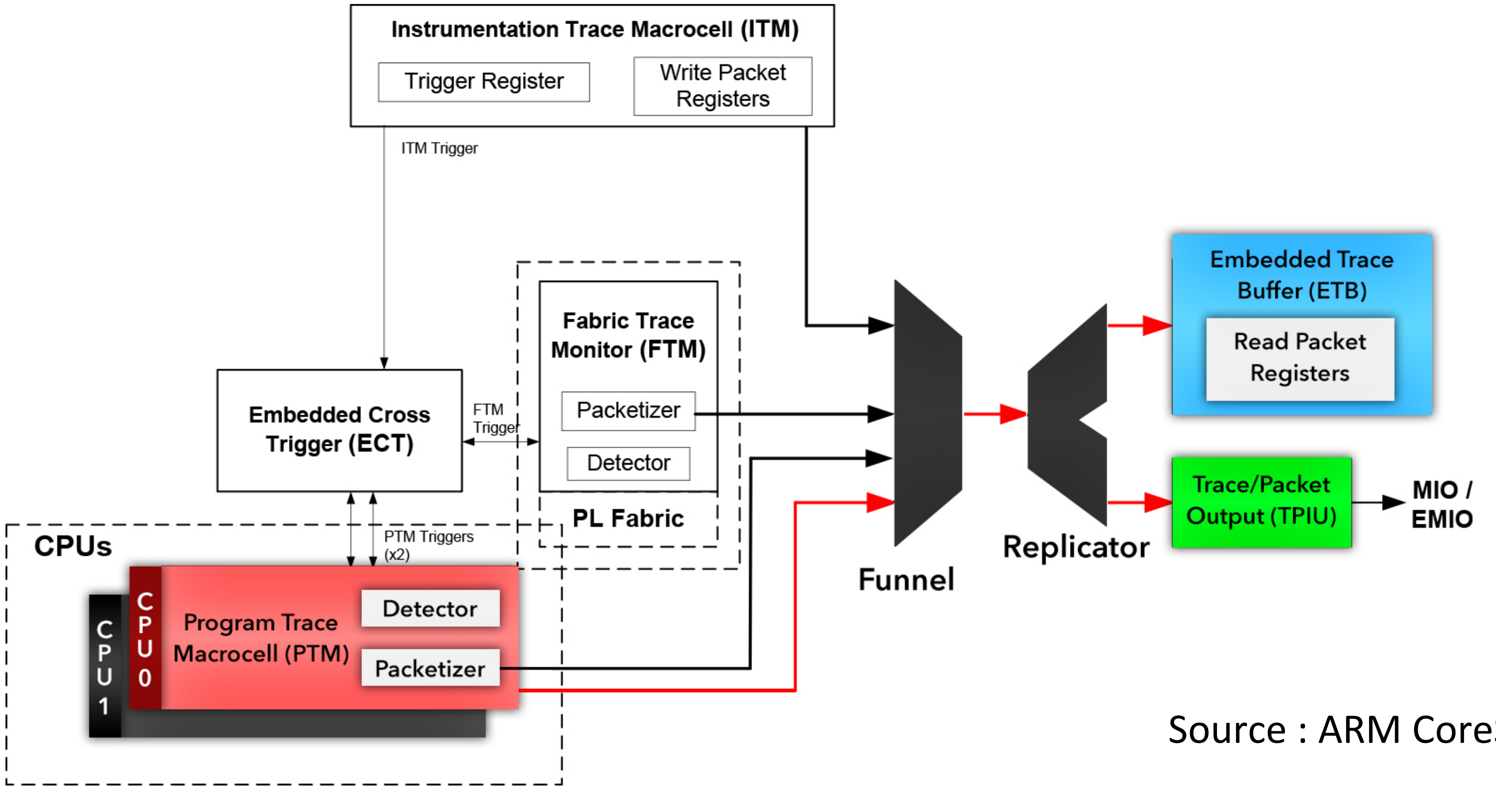

Processing System
Flash Controller NOR, NAND, SRAM, Quad SPI
Multiport DRAM Controller DDR3, DDR3L, DDR2
Processor I/O Mux
2x SPI
2x I2C
2x CAN
2x UART
GPIO
2x SDIO with DMA
2x USB with DMA
2x GigE with DMA
AMBA® Interconnect
AMBA Interconnect
ARM® CoreSight™ Multi-Core Debug and Trace
NEON™ DSP/FPU Engine
NEON DSP/FPU Engine
Cortex™- A9 MPCore 32/32 KB I/D Caches
Cortex- A9 MPCore 32/32 KB I/D Caches
512 Kbyte L2 Cache
General Interrupt Controller
Watchdog Timer
Configuration
Timers
DMA
Snoop Control Unit
256 Kbyte On-Chip Memory
AMBA Interconnect
Security AES, SHA, RSA
AMBA Interconnect
EMIO
General Purpose AXI Ports
ACP
High Performance AXI Ports
XADC 2x ADC, Mux, Thermal Sensor
Programmable Logic (System Gates, DSP, RAM)
PCIe Gen2 1-8 Lanes
Multi-Standard I/Os (3.3V & High-Speed 1.8V)
Multi-Gigabit Transceivers

Overall architecture



CoreSight components

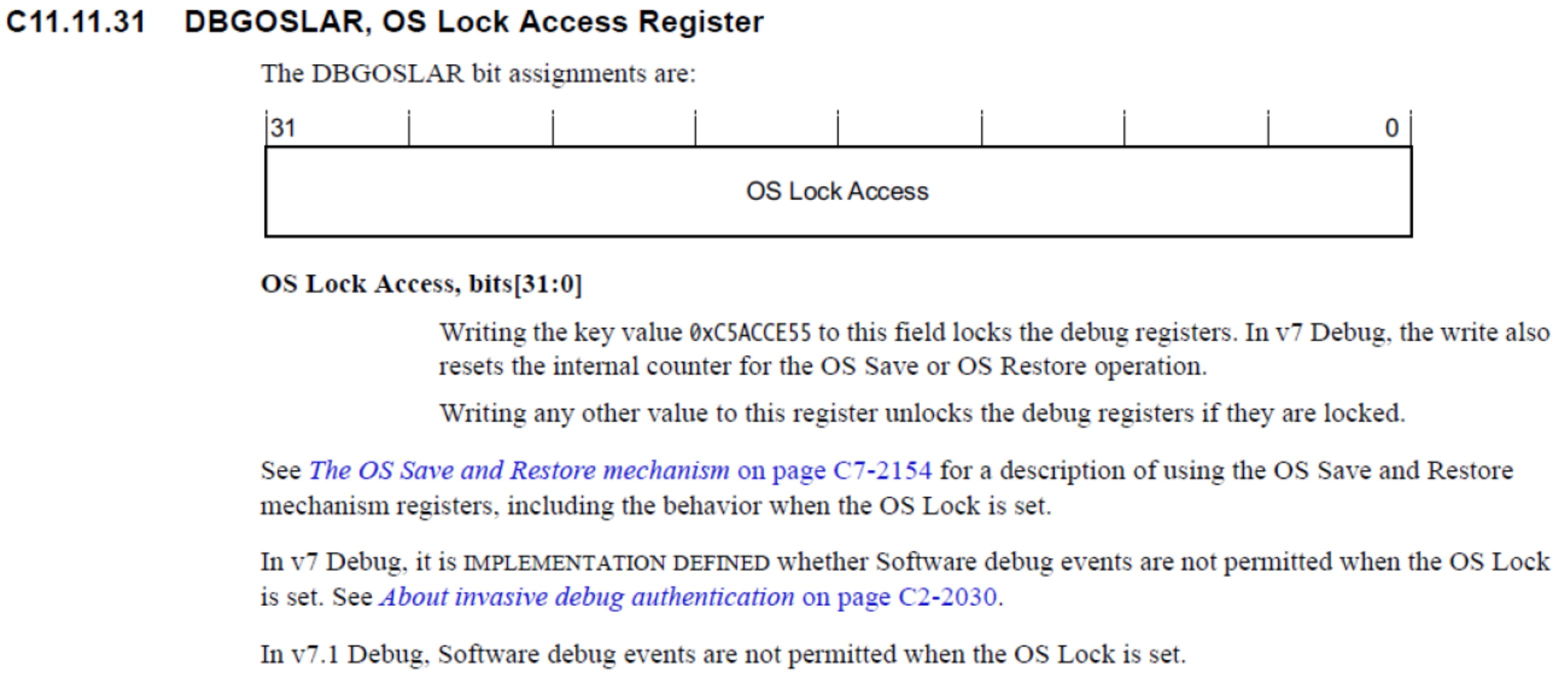
A set of IP blocks providing HW-assisted system tracing



Source : ARM CoreSight TRM

CoreSight components

A set of IP blocks providing HW-assisted system tracing



C11.11.31 DBGOSLAR, OS Lock Access Register

The DBGOSLAR bit assignments are:

31
0
OS Lock Access

OS Lock Access, bits[31:0]

Writing the key value `0xC5ACCE55` to this field locks the debug registers. In v7 Debug, the write also resets the internal counter for the OS Save or OS Restore operation.

Writing any other value to this register unlocks the debug registers if they are locked.

See *The OS Save and Restore mechanism* on page C7-2154 for a description of using the OS Save and Restore mechanism registers, including the behavior when the OS Lock is set.

In v7 Debug, it is IMPLEMENTATION DEFINED whether Software debug events are not permitted when the OS Lock is set. See *About invasive debug authentication* on page C2-2030.

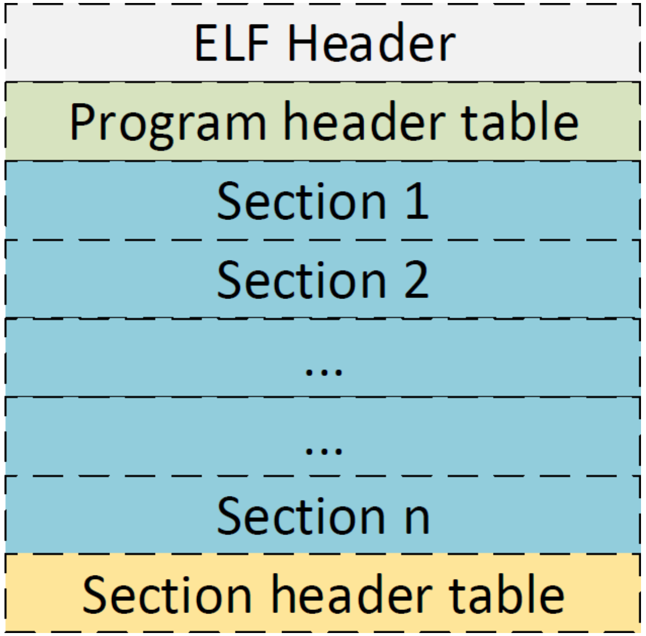
In v7.1 Debug, Software debug events are not permitted when the OS Lock is set.

Source : ARM CoreSight TRM

CoreSight PTM

Features:

- Trace filter (all code or regions of code)
- Branch Broadcast
- Context ID comparator
- Cycle accurate tracing
- Timestamping



Example trace

Source code

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

Assembly

```
8638 for_loop:
...
b 8654 :
...
866c : bcc 8654
```

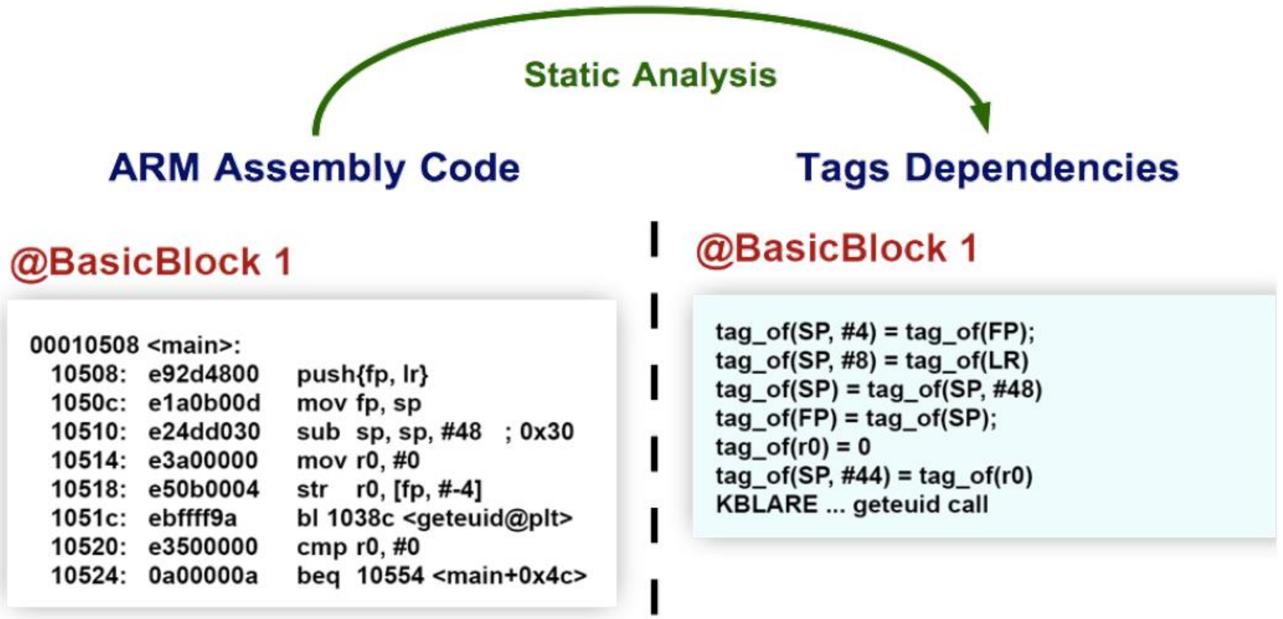
Trace

```
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)

Static analysis – Tag dependencies



```
00010508 <main>:
  10508:  e92d4800    push{fp, lr}
  1050c:  e1a0b00d    mov fp, sp
  10510:  e24dd030    sub  sp, sp, #48   ; 0x30
  10514:  e3a00000    mov r0, #0
  10518:  e50b0004    str   r0, [fp, #-4]
  1051c:  ebffff9a    bl 1038c <geteuid@plt>
  10520:  e3500000    cmp r0, #0
  10524:  0a00000a    beq  10554 <main+0x4c>
```

```
tag_of(SP, #4) = tag_of(FP);
tag_of(SP, #8) = tag_of(LR)
tag_of(SP) = tag_of(SP, #48)
tag_of(FP) = tag_of(SP);
tag_of(r0) = 0
tag_of(SP, #44) = tag_of(r0)
KBLARE ... geteuid call
```

DIFT stack

Our case:

- We want to store tags and initialize tags from the operating system : modified **kBlare** (based on a Linux Kernel 4.9)

- We don’t want to loose information (no over-approximation) : **Dynamic approach:** Instrumentation + PTM traces

- Extract some informations about the data flow (for tag propagation) : **Static Analysis:** Generating annotations.

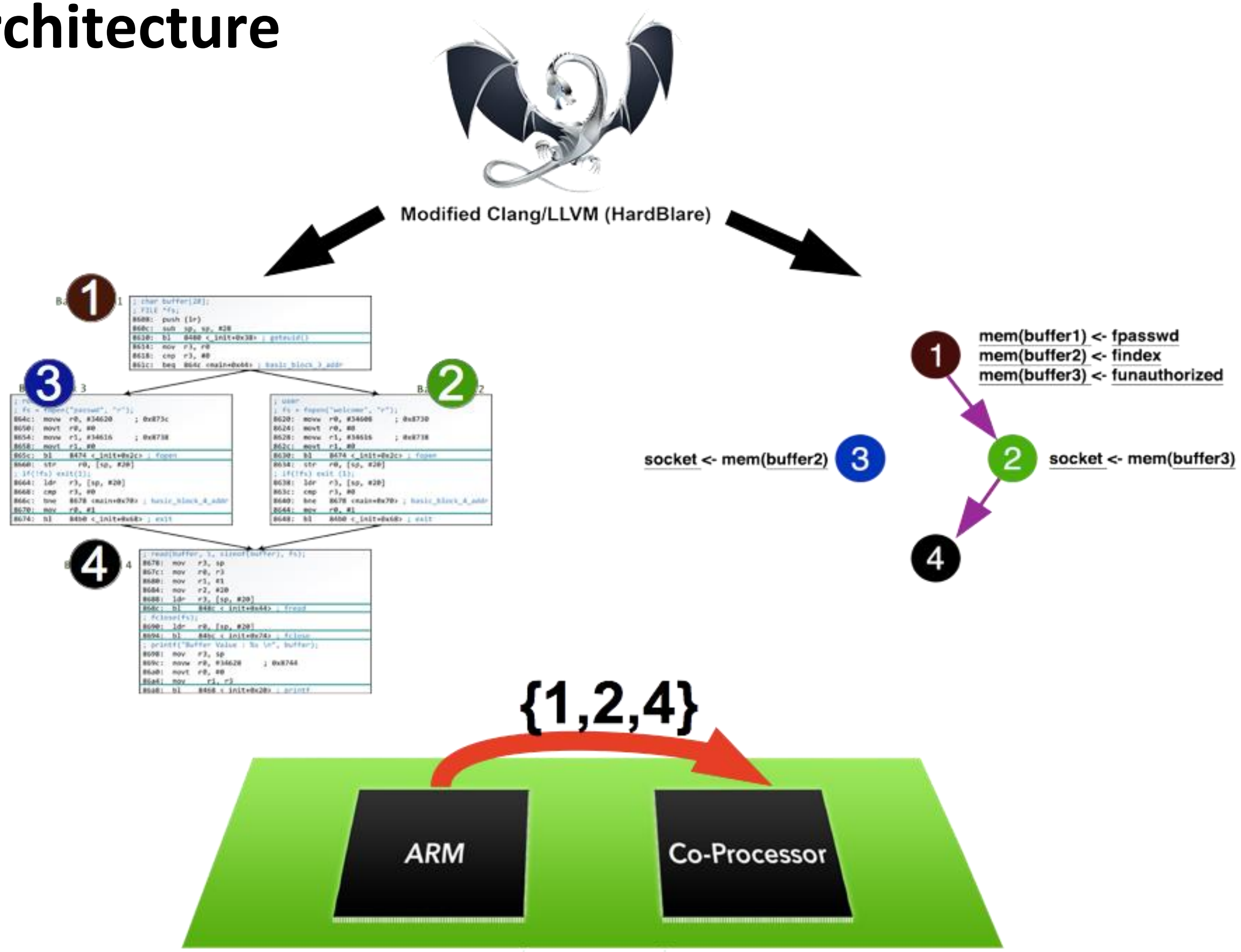
Overall architecture

```
char buffer1[20], buffer2[20], buffer3[20];
FILE *fpassword, findex, funauthorized;

fpassword = open("passwd.txt");
findex = open("index.html");
funauthorized = open("unauthorized.html");

read(buffer1, fpassword)
read(buffer2, findex)
read(buffer3, funauthorized)

if(getuid()){
    send_to_socket(buffer2);
}
else{
    send_to_socket(buffer3);
}
```

Modified Clang/LLVM

- **LLVM modification:**
  - Static Analysis
  - Generating annotations
  - Code instrumentation
  - Modifying the ARM Backend (excl. conditional exec, ...)
- **kBlare:** loading annotations into the co-processor dedicated memory.
- **Dynamic linker:** resolving addresses of instrumentation IP address, merging annotations.
- **Yocto:** Generating the complete distribution and the SDK suitable for HardBlare.

Overall architecture



```
; char buffer[20];
; FILE *fs;
8608:  push {lr}
860c:  sub  sp, sp, #28
8610:  bl   8400 <_init+0x38> ; geteuid()
8614:  mov  r3, r0
8618:  cmp  r3, #0
861c:  beq  864c <main+0x44> ; basic_block_3_addr
```

```
; r...
; fs = fopen("passwd", "r");
864c:  movw  r0, #34620     ; 0x873c
8650:  movt  r0, #0
8654:  movw  r1, #34616     ; 0x8738
8658:  movt  r1, #0
865c:  bl    8474 <_init+0x2c> ; fopen
8660:  str     r0, [sp, #20]
; if(!fs) exit(1);
8664:  ldr   r3, [sp, #20]
8668:  cmp   r3, #0
866c:  bne   8678 <main+0x70> ; basic_block_4_addr
8670:  mov   r0, #1
8674:  bl    84b0 <_init+0x68> ; exit
```

```
; user
; fs = fopen("welcome", "r");
8620:  movw  r0, #34608     ; 0x8730
8624:  movt  r0, #0
8628:  movw  r1, #34616     ; 0x8738
862c:  movt  r1, #0
8630:  bl    8474 <_init+0x2c> ; fopen
8634:  str   r0, [sp, #20]
; if(!fs) exit (1);
8638:  ldr   r3, [sp, #20]
863c:  cmp   r3, #0
8640:  bne   8678 <main+0x70> ; basic_block_4_addr
8644:  mov   r0, #1
8648:  bl    84b0 <_init+0x68> ; exit
```

```
; read(buffer, 1, sizeof(buffer), fs);
8678:  mov   r3, sp
867c:  mov   r0, r3
8680:  mov   r1, #1
8684:  mov   r2, #20
8688:  ldr   r3, [sp, #20]
868c:  bl    848c <_init+0x44> ; fread
; fclose(fs);
8690:  ldr   r0, [sp, #20]
8694:  bl    84bc <_init+0x74> ; fclose
; printf("Buffer Value : %s \n", buffer);
8698:  mov   r3, sp
869c:  movw  r0, #34620     ; 0x8744
86a0:  movt  r0, #0
86a4:  mov     r1, r3
86a8:  bl    8458 <_init+0x20> ; printf
```

Instrumentation

Recover memory addresses

Two possible strategies:

- Recover all memory addresses through instrumentation
- Recover only register-relative memory address through instrumentation

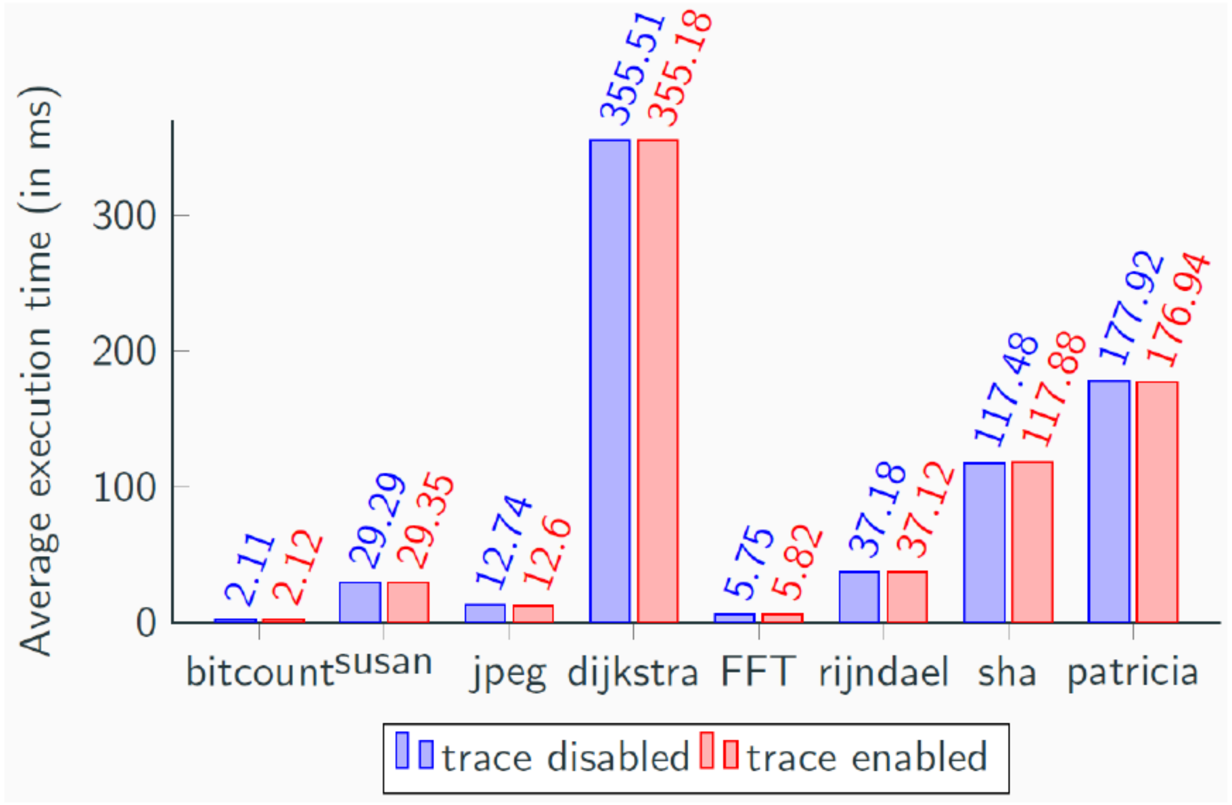
Instrumentation strategy 1

Example Instructions	Tag dependencies	Memory address recovery
`sub r0, r1, r2`	$\underline{\texttt{r0}} = \underline{\texttt{r1}} + \underline{\texttt{r2}}$	
`mov r3, r0`	$\underline{\texttt{r3}} = \underline{\texttt{r0}}$	
`str r1, [PC, #4]`	$\underline{\texttt{@Mem(PC+4)}} = \underline{\texttt{r1}}$	instrumented
`ldr r3, [SP, #-8]`	$\underline{\texttt{r3}} = \underline{\texttt{@Mem(SP-8)}}$	instrumented
`str r1, [r3, r2]`	$\underline{\texttt{@Mem(r3+r2)}} = \underline{\texttt{r1}}$	instrumented

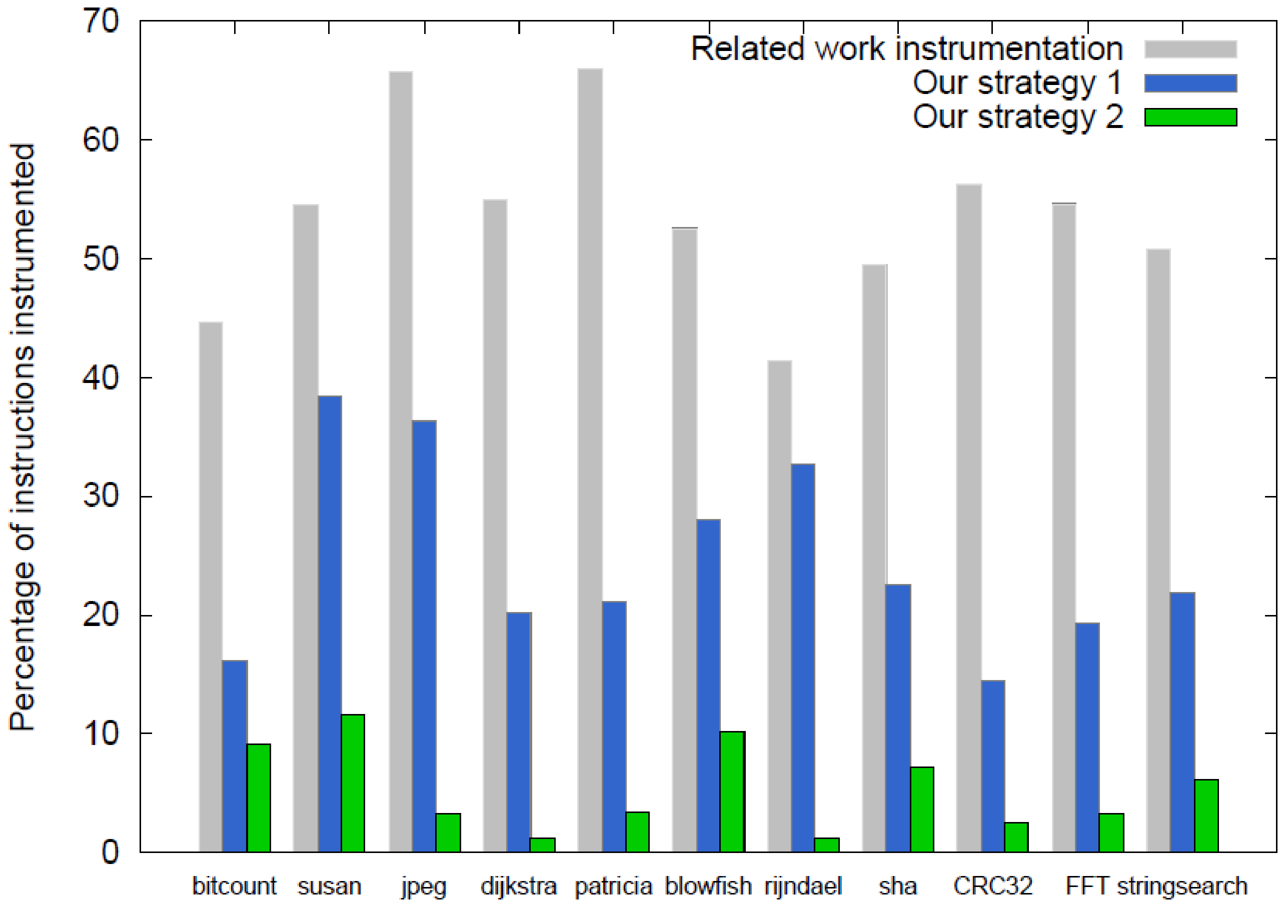
Instrumentation strategy 2

Example Instructions	Tag dependencies	Memory address recovery
`sub r0, r1, r2`	$\underline{\texttt{r0}} = \underline{\texttt{r1}} + \underline{\texttt{r2}}$	
`mov r3, r0`	$\underline{\texttt{r3}} = \underline{\texttt{r0}}$	
`str r1, [PC, #4]`	$\underline{\texttt{@Mem(PC+4)}} = \underline{\texttt{r1}}$	CoreSight PTM
`ldr r3, [SP, #-8]`	$\underline{\texttt{r3}} = \underline{\texttt{@Mem(SP-8)}}$	Static analysis
`str r1, [r3, r2]`	$\underline{\texttt{@Mem(r3+r2)}} = \underline{\texttt{r1}}$	instrumented

CoreSight components – Performance overhead



Instrumentation time overhead



Comparison with existing works – Hardware view

Approaches	**Kannan**	**Deng**	**Heo**	**ARMHEx**
Hardcore portability	No	No	**Yes**	**Yes**
Main CPU	Softcore	Softcore	Softcore	**Hardcore**
Communication overhead	N/A	N/A	60%	**5.4%**
Area overhead	6.4%	14.8%	14.47%	**0.47%**
Area (Gate Counts)	N/A	N/A	256177	**128496**
Power overhead	N/A	**6.3%**	24%	16%
Max frequency	N/A	**256 MHz**	N/A	250 MHz
Isolation	No	No	No	**Yes**

Conclusion

As well as some perspectives ☺

Conclusion - Perspectives

Take away:

- CoreSight PTM allows to obtain runtime information (Program Flow)
- Non-intrusive tracing => Negligible performance overhead

Perspectives:

- Mid-term: releasing a PoC of the whole system
- Combine Low-level and OS-level DIFT
- Extend DIFT on multicore CPU
- Take use of other debug components for security
- What about Intel, ST ?

Thanks!

Any questions?

You can find me at: pascal.cotret@gmail.com