Monitoring program execution (and more!) on ARM processors

Toulouse Hacking Convention_

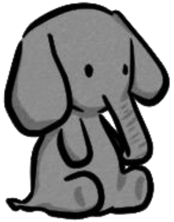
Pascal Cotret
pascal.cotret@gmail.com / @Pascal_r2

N7, March 9, 2018

Hello!

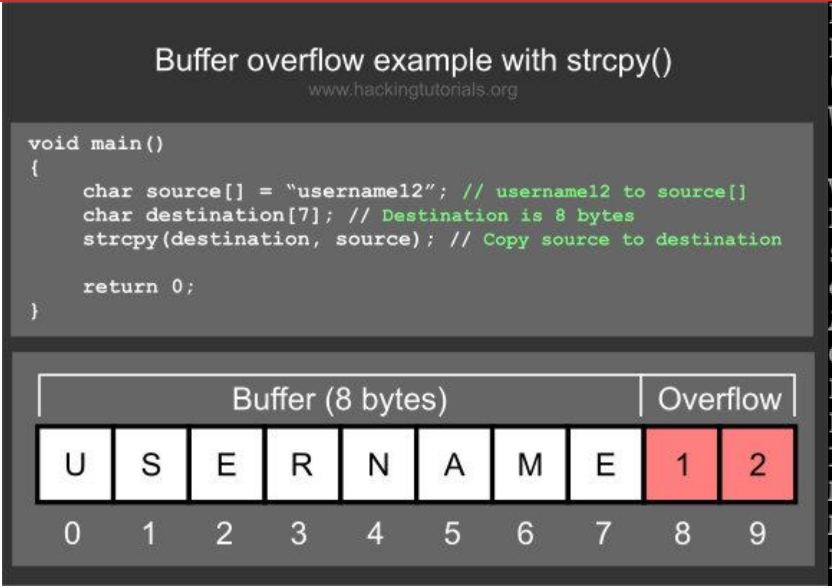
- Embedded software security engineer
- Researcher in my spare-time

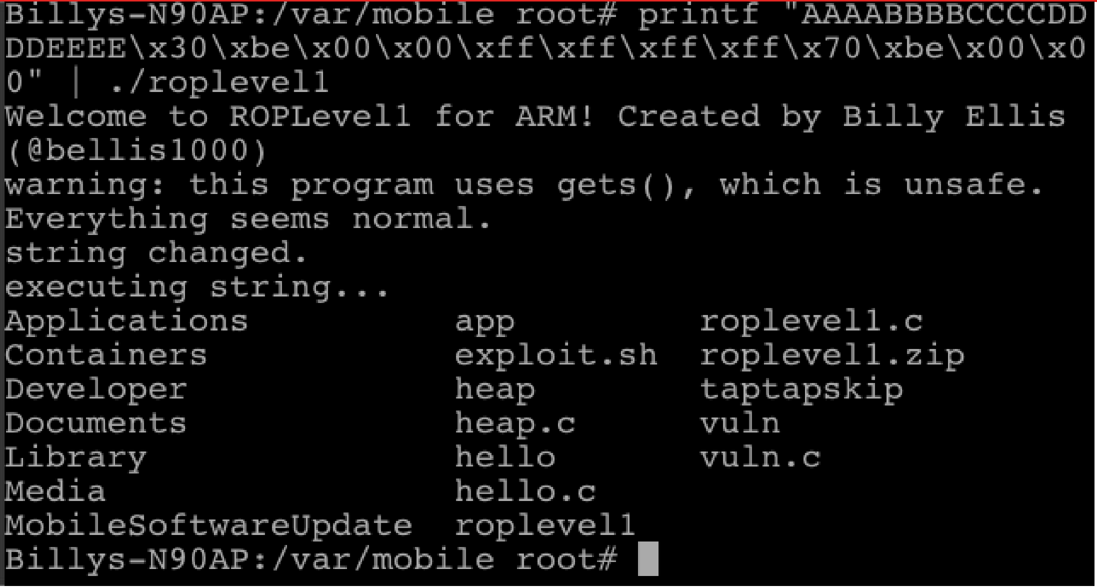
(also former associate professor)



HardBlare project (3 labs, 2 PhDs…)

Threat model





Playing with such attacks on ARM:

https://billy-ellis.github.io (@bellis1000)

https://www.root-me.org/?page=recherche&lang=en&recherche=ARM

https://azeria-labs.com/ (@Fox0x01)

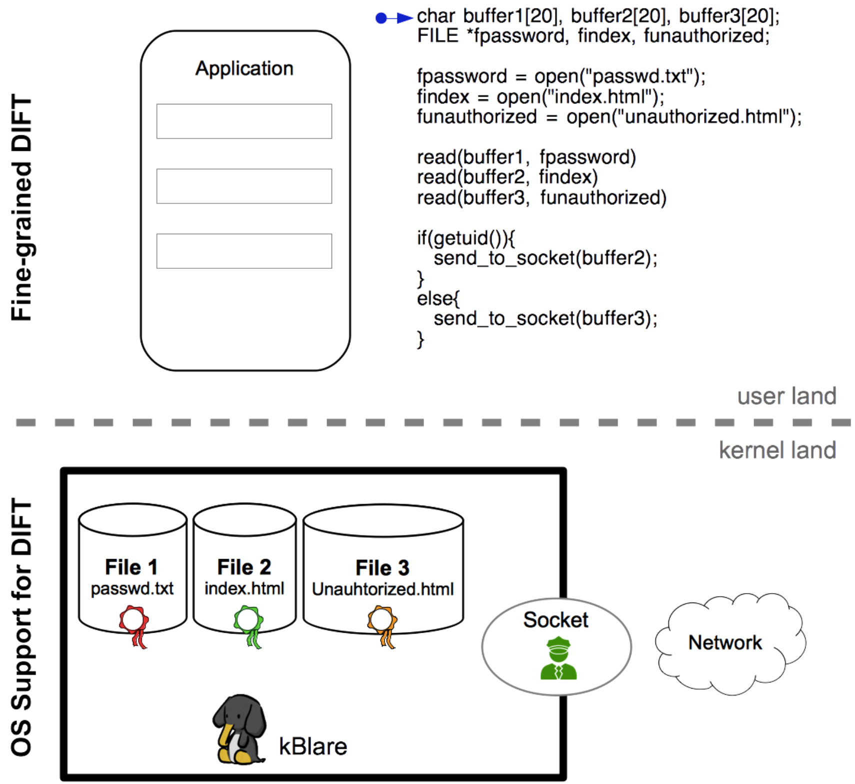
DIFT = Dynamic Information Flow Tracking

- DIFT => Detection of software attacks
  - Buffer overflow, Return Oriented Programming, etc.

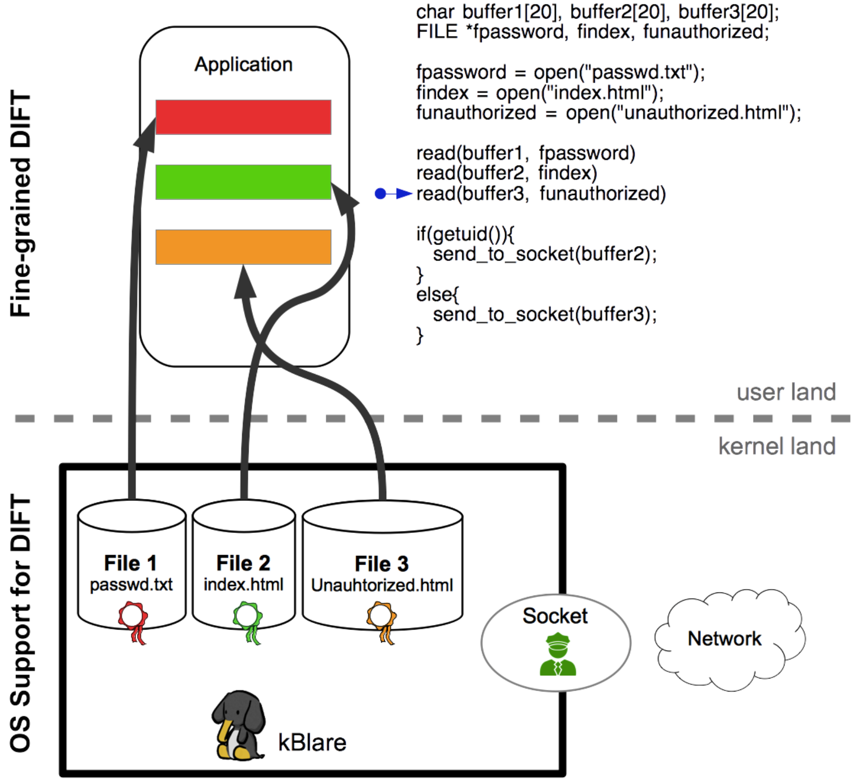
- Security purposes => Integrity and Confidentiality

- Principle:
  - Tags attached to containers + relationship
  - At runtime, propagate tags
  - Detecting any violation at run-time asap

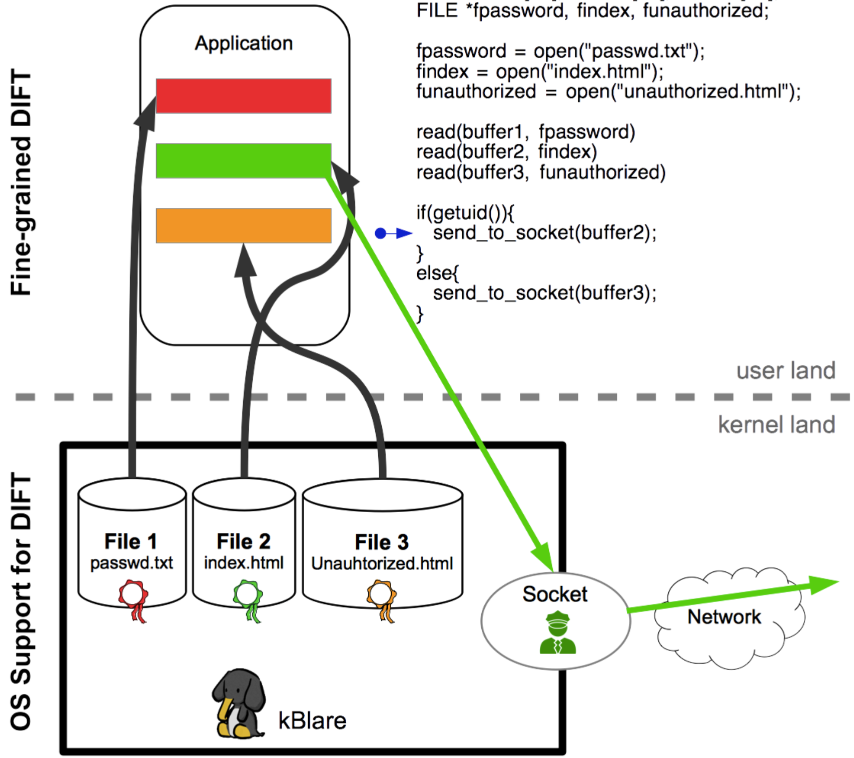
DIFT = Dynamic Information Flow Tracking



DIFT = Dynamic Information Flow Tracking



DIFT = Dynamic Information Flow Tracking



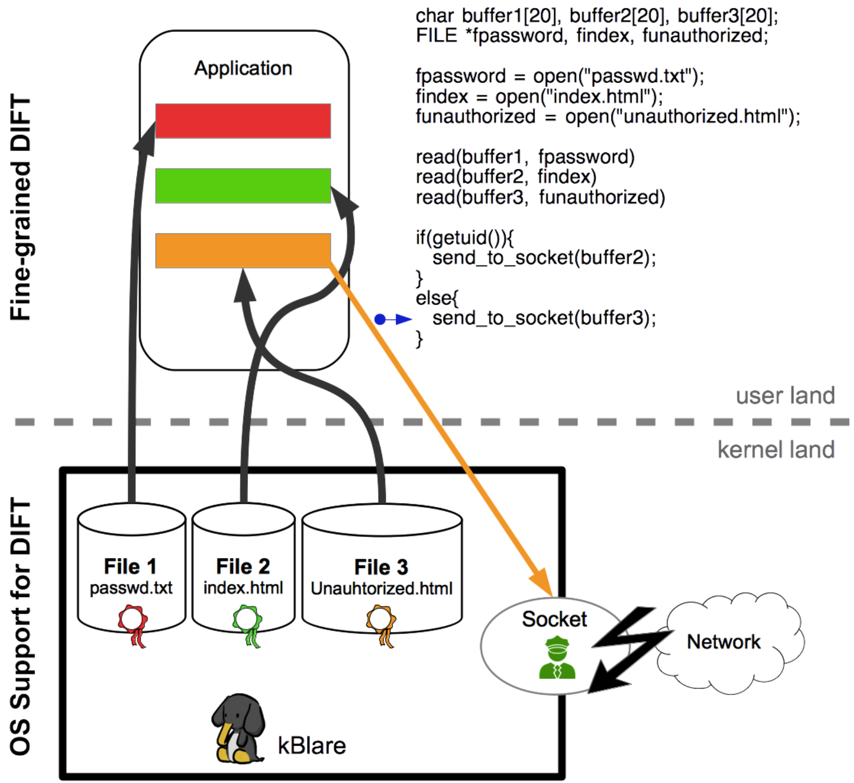

```
char buffer1[20], buffer2[20], buffer3[20];
FILE *fpassword, findex, funauthorized;

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 = Dynamic Information Flow Tracking



Different levels for DIFT

- Operating system:

Files / Executables

- Language level:

Variables / Functions

- Processor level:

Address, registers / Instructions

DIFT – Memory corruption detection

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 – Memory corruption detection

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 – Memory corruption detection

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 – Memory corruption detection

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]

Attacker overwrites return address and takes control

```c
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]

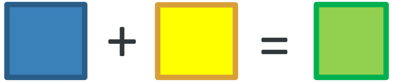
Different levels for DIFT

- 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

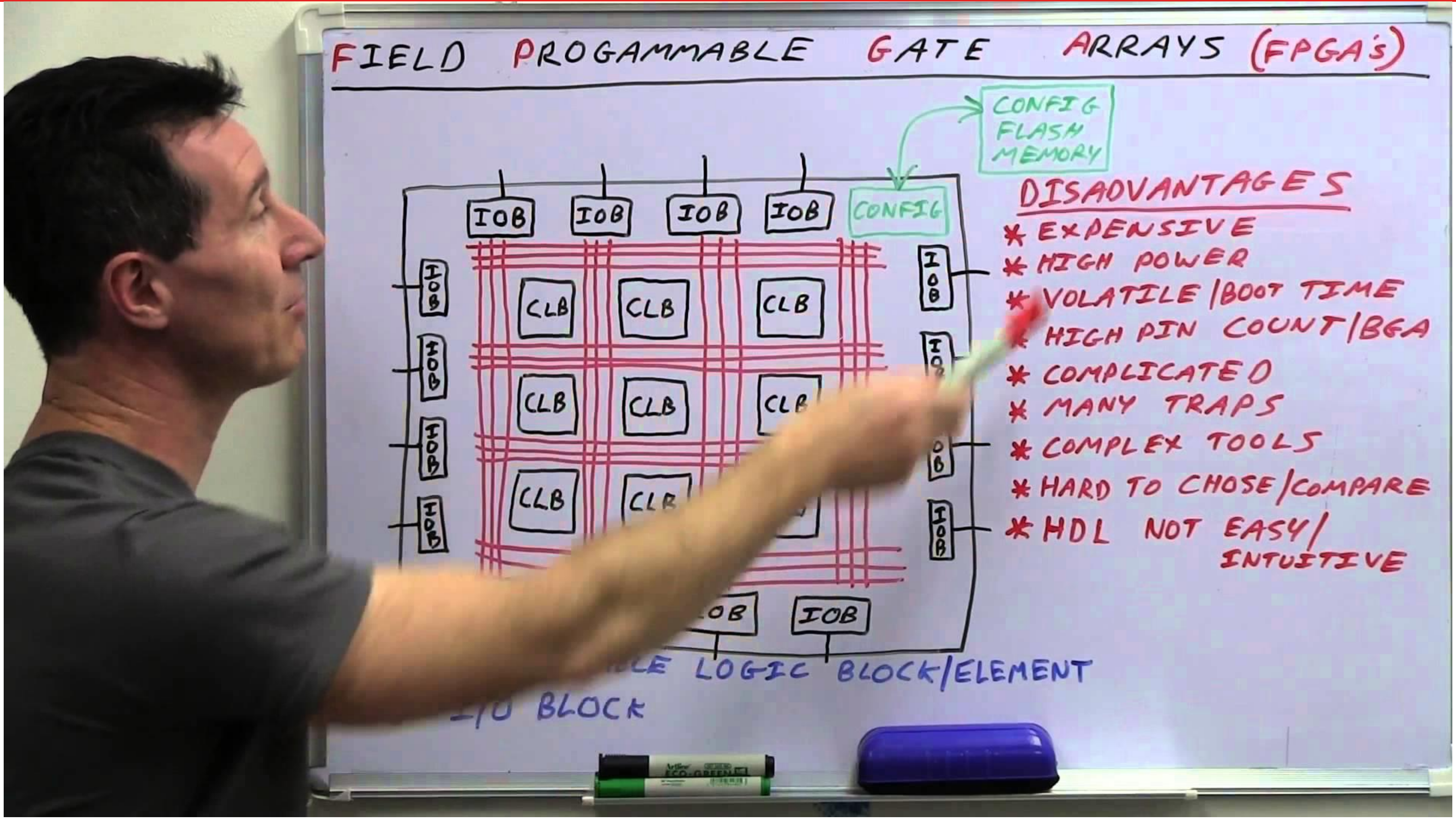
- Application level
  - Java / Android, Javascript, C
- OS level
  - kBlare (Linux kernel w/ software IFT)
- Low level
  - Deeping into processor architecture maybe?

Different levels for DIFT

- Application level
  - Java / Android, Javascript, C
- OS level
  - kBlare (Linux kernel w/ software IFT)
- Low level
  - Deeping into processor architecture maybe?

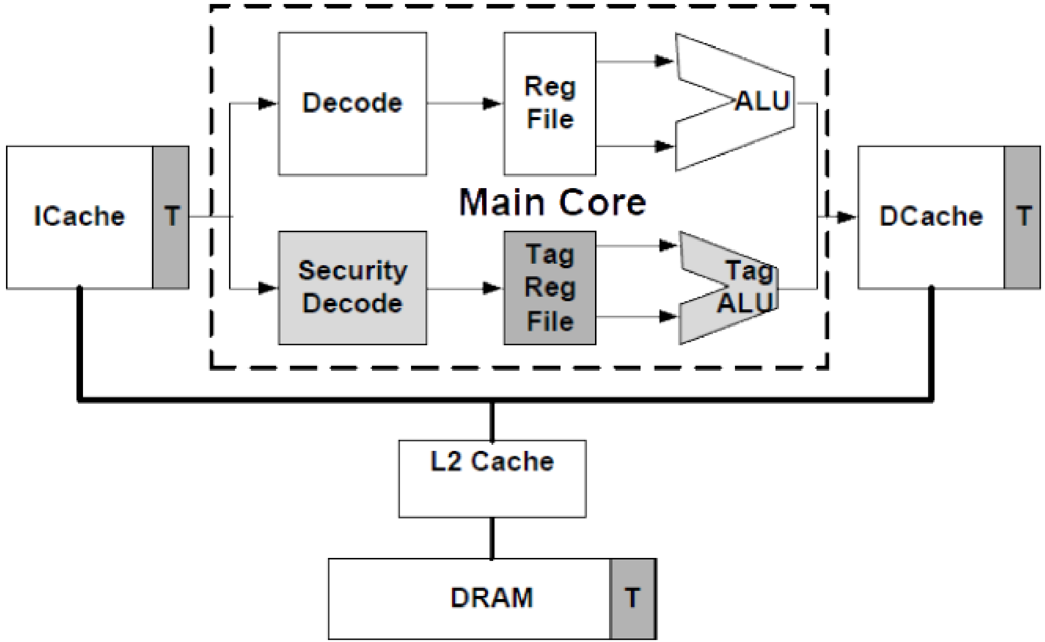
Buying an ARM license => no way. Or...

FPGA => Programmable electronics

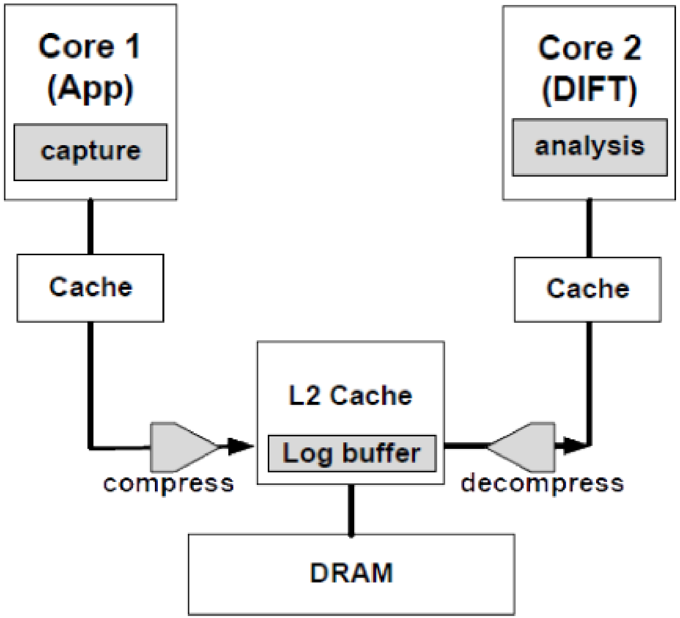


Source: EEVBlog #496 – What is an FPGA? (Youtube)

Different levels for DIFT

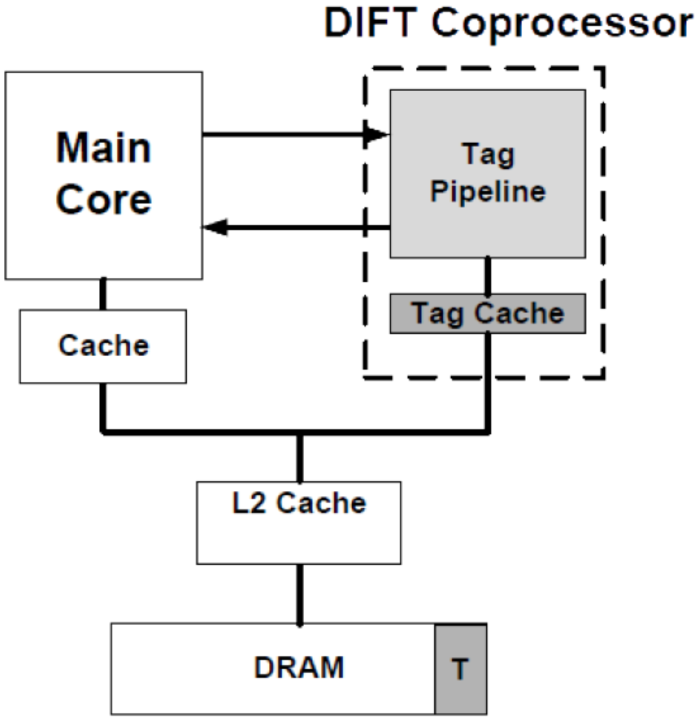


In-core DIFT



Offloading

Different levels for DIFT



Off-core DIFT

Related works

	Advantages	Disadvantages
Software	Flexible security policies	Overhead (300% at least…)
In-core DIFT	Low overhead (10%)	Invasive modifications
Dedicated CPU	Low overhead (10%)	Wasting resources
Dedicated coprocessor	Low overhead (10%) CPU not modified	CPU/coprocessor communication

ARMHEx approach

- Limiting the impact of software instrumentation

- Security of the coprocessor

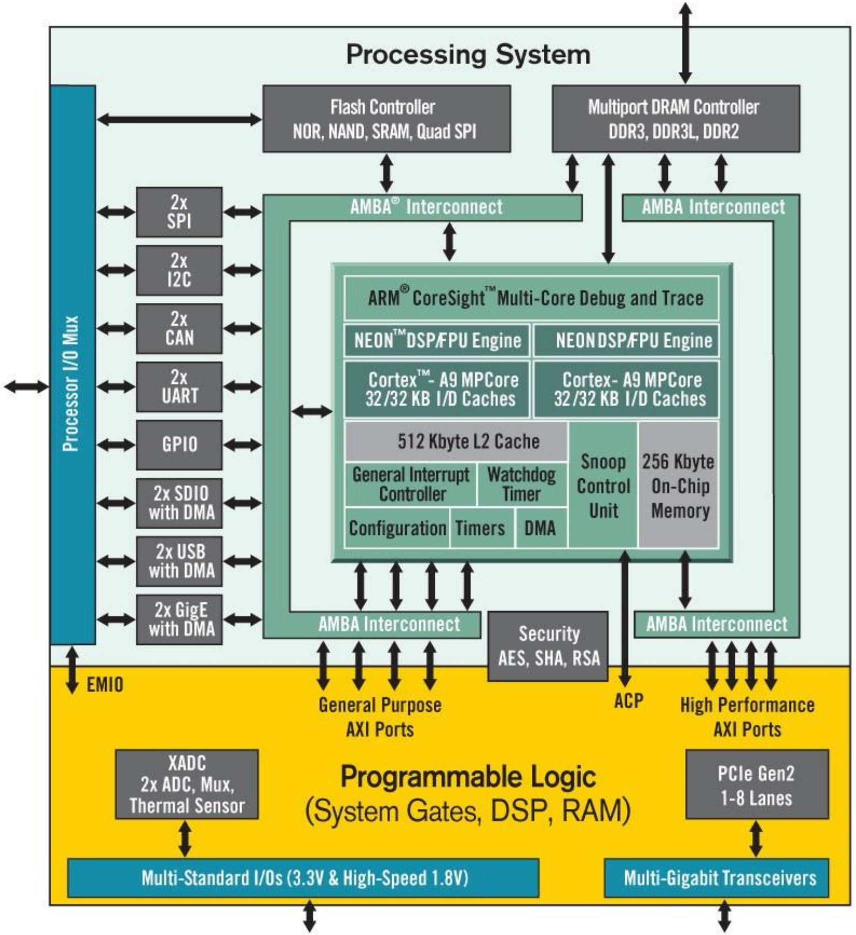
- First work on ARM-based SoCs

- Additional challenges

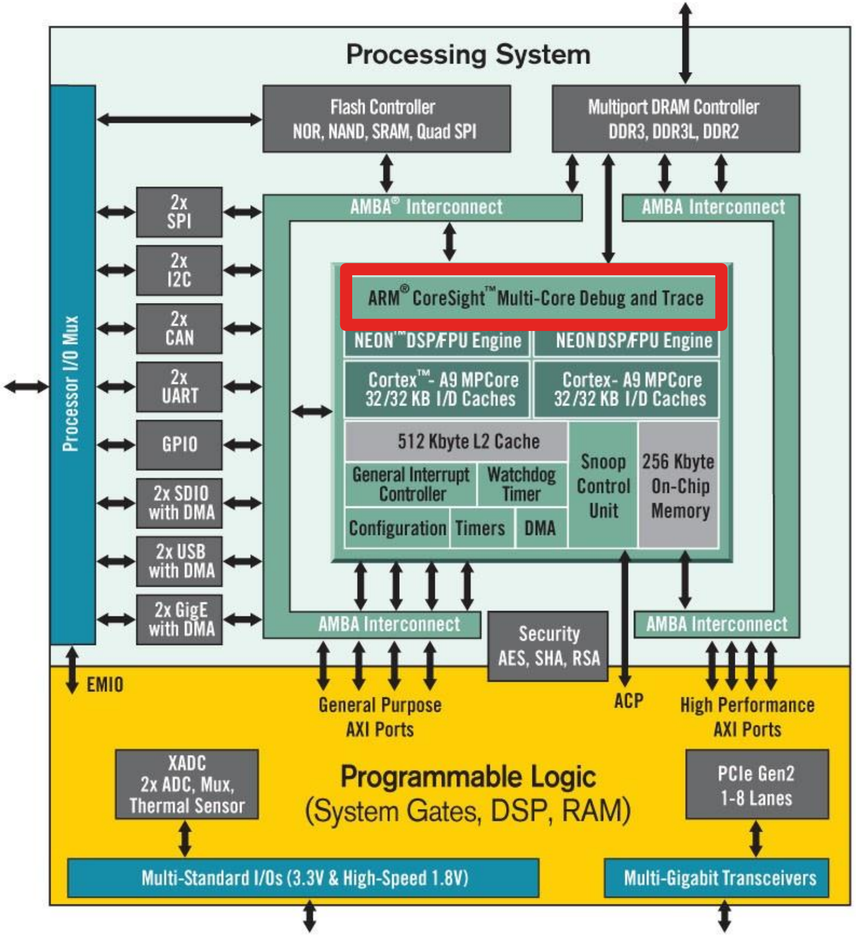
ARMHEx approach

- Limiting the impact of software instrumentation
- Security of the coprocessor
- **First work on ARM-based SoCs**
- Additional challenges

What can I do with my processor?

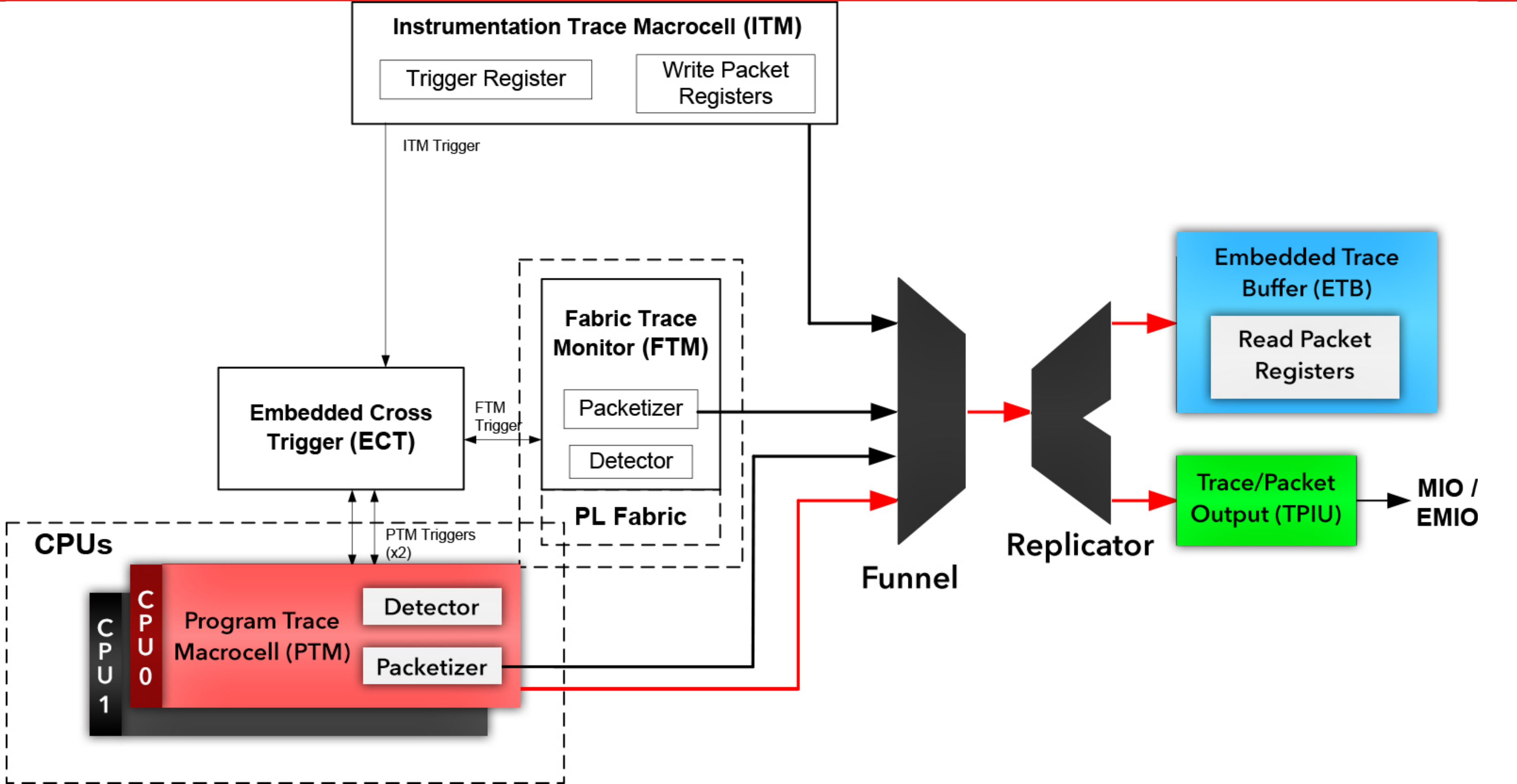


What can I do with my processor?

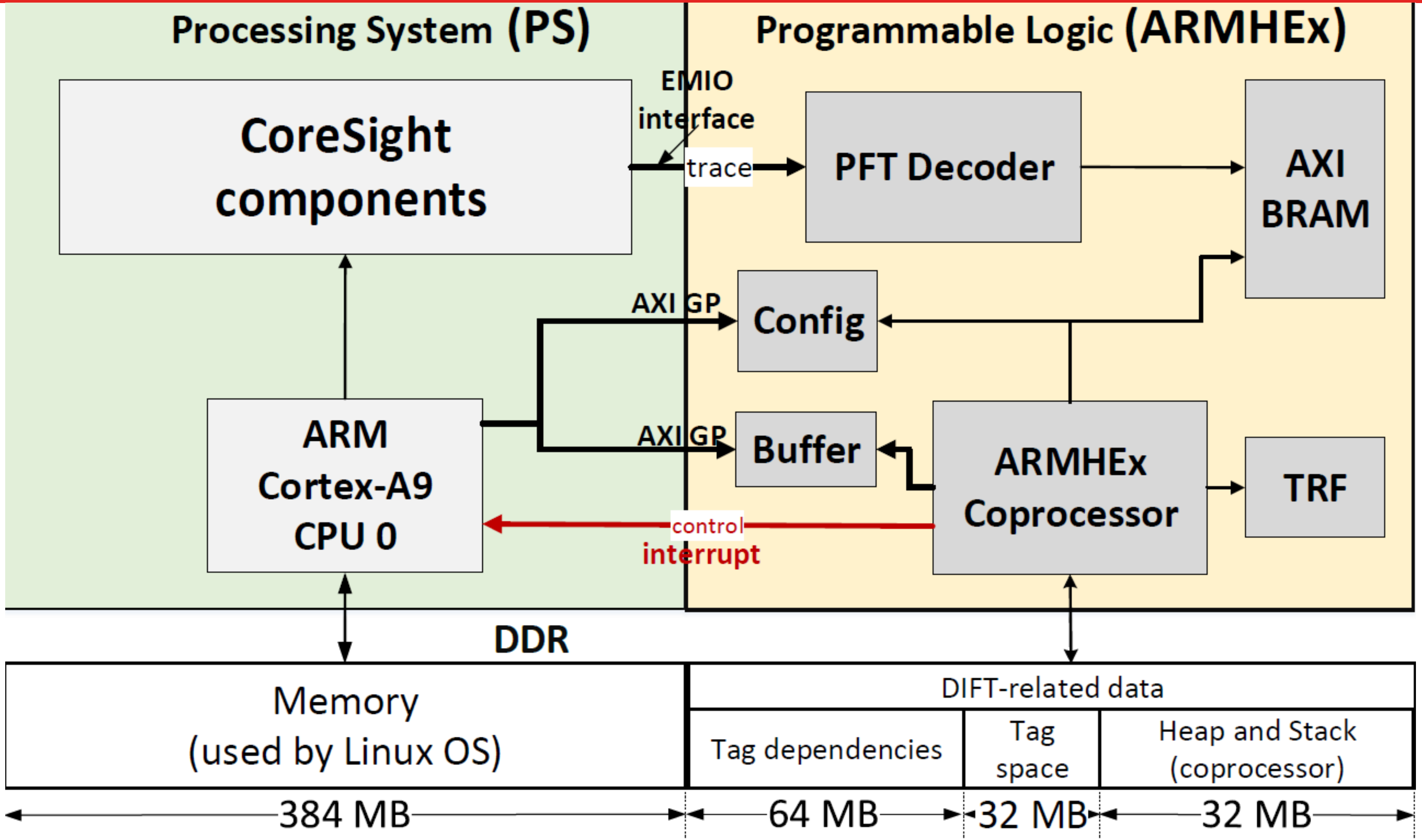


- CoreSight: debug components
- Available in most of Cortex-A + Cortex-M3 (for ARM)
- Can export stuff

CoreSight components



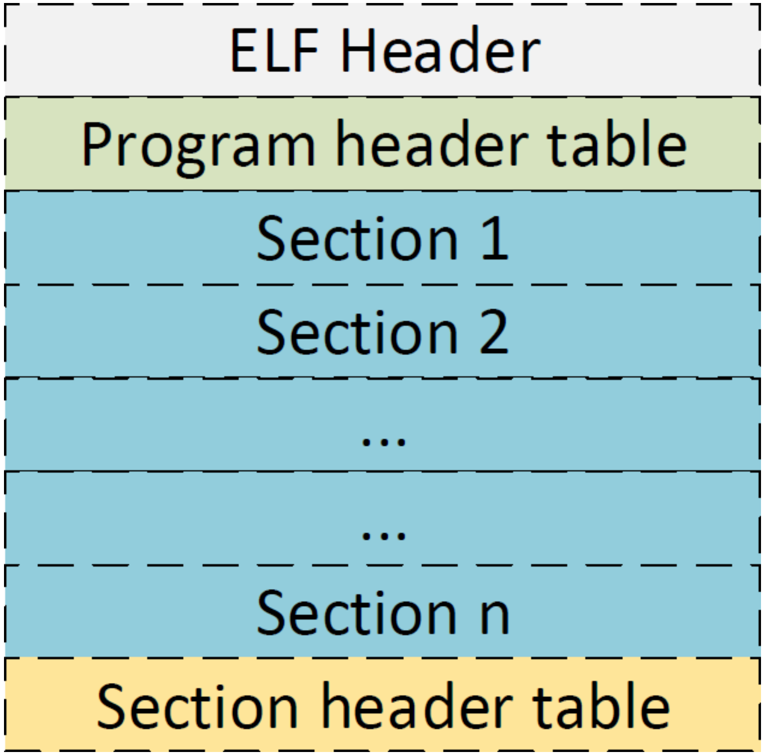
CoreSight components – Where should I export my metadata?



CoreSight PTM features

Features:

- Trace filter
- Branch Broadcast
- Timestamping
- Etc, etc.



What does a trace look like?

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

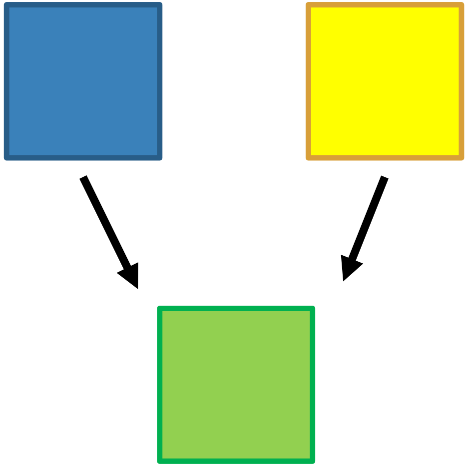
A-sync
Address 00008638, (I-sync Context 00000000, IB 21)
Address 00008654, Branch Address packet (x 10)

DIFT toolchain

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.

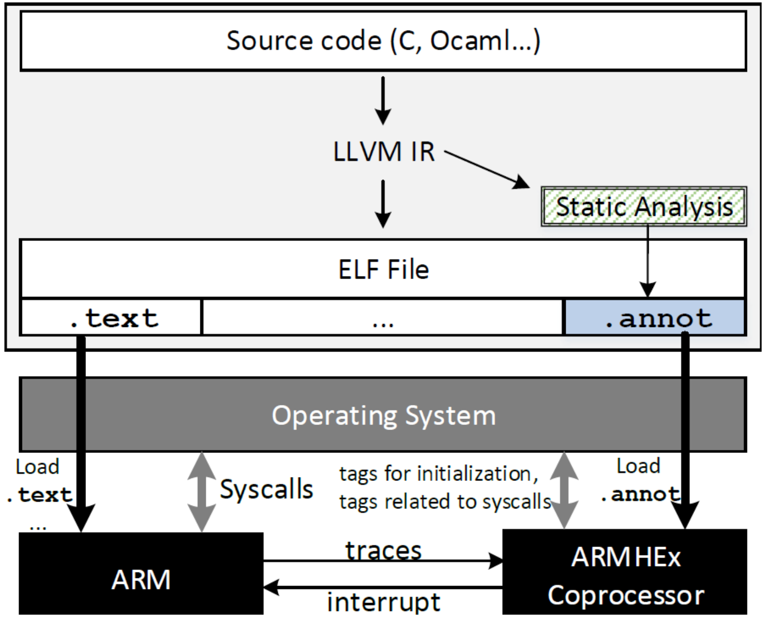
Generating annotations



(status on late February)

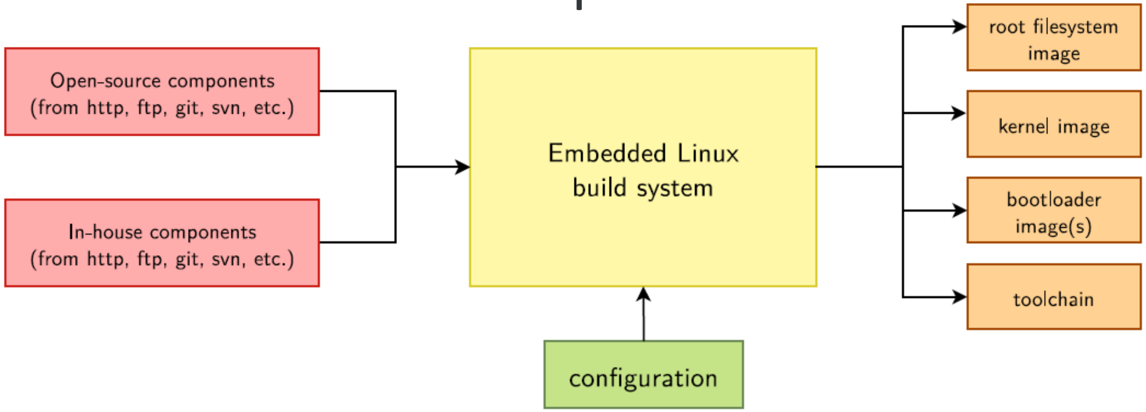
- 200 instructions done:
  - LLVM meta-instructions
  - « Basic » stuff: add, compare, load/store, etc.
- TODO: 200 instructions left (at least…)
  - Parallel additions/substractions features
  - Advanced SIMD instructions

DIFT toolchain



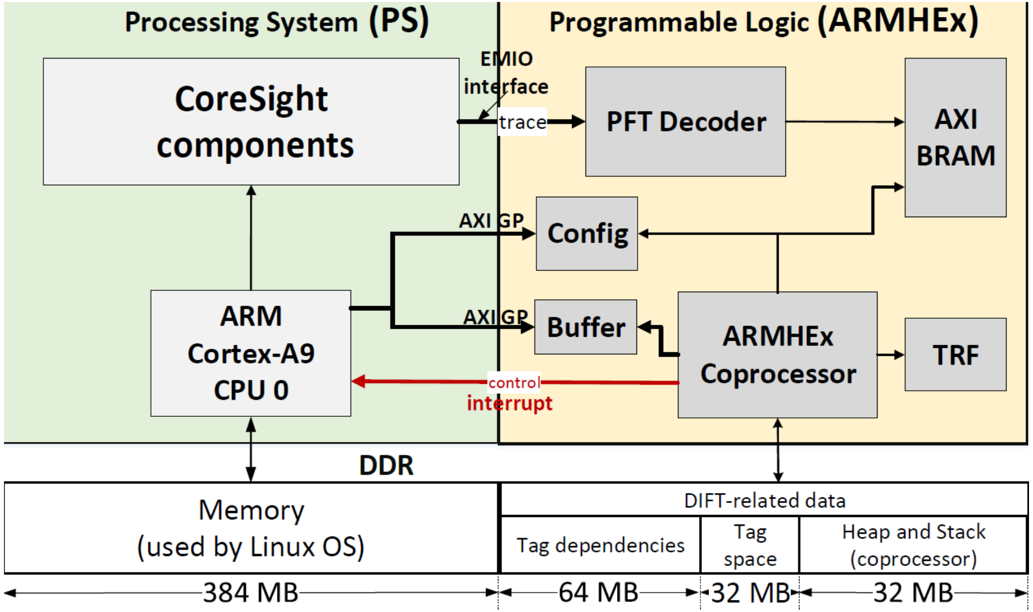
yocto PROJECT

- Templates/tools/methods
- Custom embedded Linux
- HardBlare recipes added



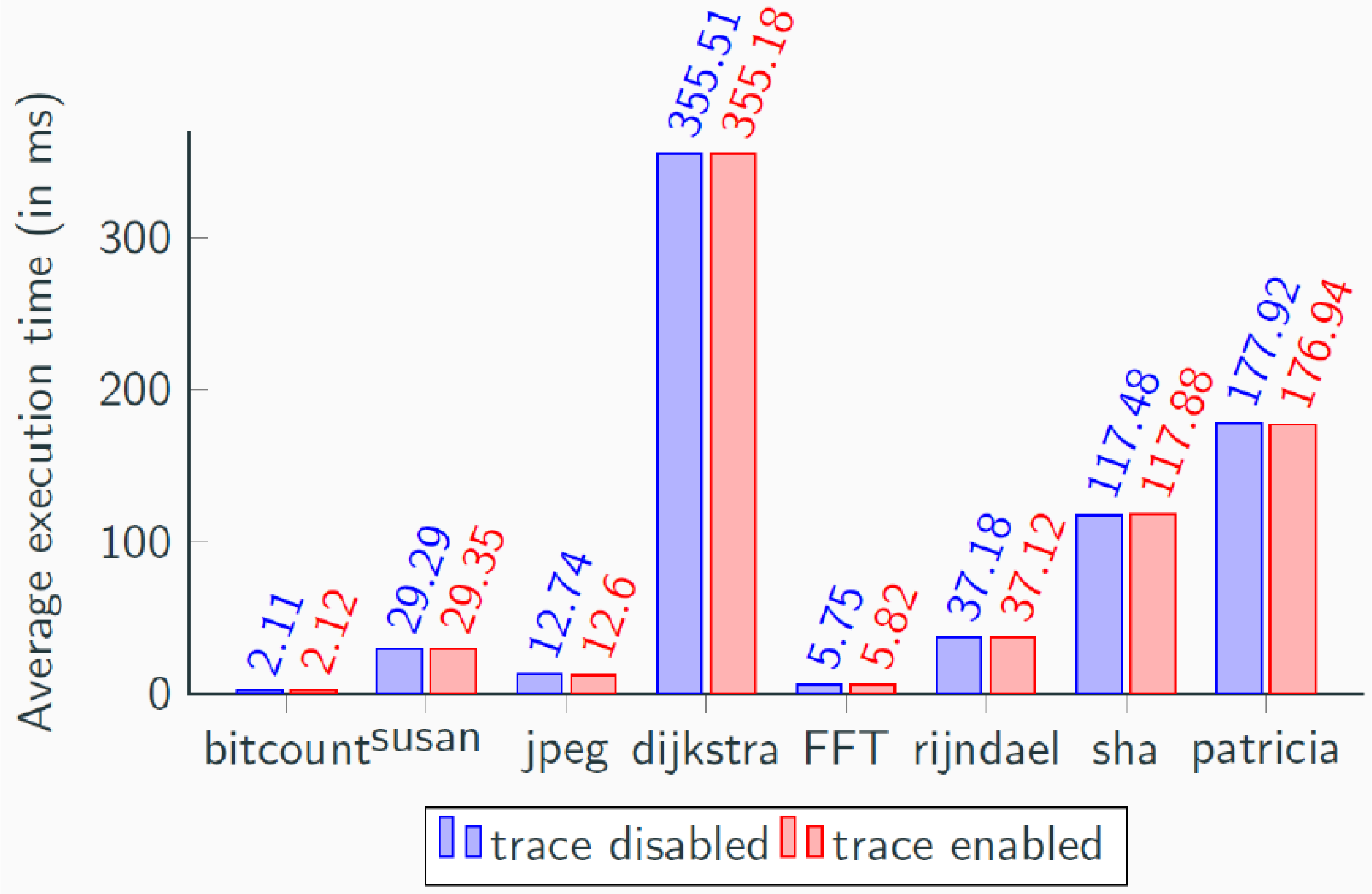
Source: Bootlin (aka Free Electrons)

Coprocessor – Quick hints



- DIFT metadata protection
  - TrustZone + secure world
- Main challenge: speed!

Some latency results



Comparison w/ existing works

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**

Perspectives

Take away:

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

RaspberryPi PoC (hopefully March)

Full PoC later this year (SoC files + Yocto)

Intel / ST? (study)

Multicore multi-thread IFT

Full-speed IFT

Monitoring program execution (and more!) on ARM processors

Toulouse
Hacking
Convention_

Pascal Cotret
pascal.cotret@gmail.com / @Pascal_r2

Many thanks to Muhammad, Mounir, Guy, Guillaume, Vianney and Arnab