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

Guillaume Hiet and partners

February 18, 2020

CIDRE research group

11 permanent researchers, 3 Post-doc, 12 PhD students

`https://team.inria.fr/cidre/`

Attack comprehension

- Hardware attacks (side channel, fault injection)
- Malware analysis (Android & Windows)

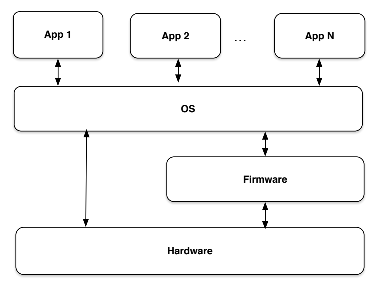
Attack detection (anomaly-based intrusion detection)

- **Low-level software (OS, firmware)**
- Distributed systems (cloud, **Industrial Control Systems**, etc.)
- Detection of ransomware attacks

Attack resistance

- **Formal methods for security**
- Deceptive security
- Blockchain

Low Level Components



Hardware-based Security Mechanisms

- Rely on hardware mechanisms (*e.g.* CPU rings, SMM, etc.)
- Used by trusted software to protect from non-trusted code

Security of Software/Hardware Interfaces

Characteristics of HSM

- Security mechanisms implemented in hardware $\rightarrow$ more secure, lower runtime overhead
- Complexe interactions with other software and hardware components $\rightarrow$ potential vulnerabilities

Research Tracks

- Can we trust existing HSM (*e.g.* SMM, SGX, TrustZone, etc.)?
  - SpecCert: Specifying and Verifying Hardware-based Security Enforcement
  - FreeSpec: Modular Verification of Components
- Can we propose new HSM?
  - Collaboration with HP Labs: Co-processor-based Behavior Monitoring of SMM Code
  - HardBlare: an Efficient Hardware-assisted DIFC for Non-modified Embedded Processors

HardBlare project

General information

- Started in October 2015. Duration: 3 years (some works are still ongoing)
- Funding: 2 PhD students and 1 PostDoc

Partners

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

How to secure embedded systems?

- The best strategy would be to avoid vulnerabilities
- Indeed many **preventive approaches** have been proposed
  - Static analysis of software code
  - Dynamic verification enforced by the runtime environment
  - Cryptography, etc.
- In practice
  - Preventive approaches are not systematically used (*e.g.* a lot of software are still using C)
  - They are not sufficient to prevent all the attacks (*e.g.* using Java or OCaml does not prevent logical errors)
- It is also important to **monitor** systems to **detect intrusions at runtime**
- Detecting attacks or intrusions is just the first step of reactive security and alerts could be used to
  - Notice security incidents to administrators
  - Stop or modify execution
  - Put the system in quarantine, etc.

Dynamic Information Flow Tracking

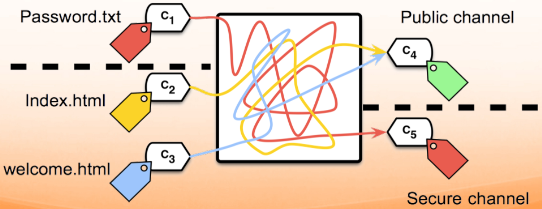
CominLabs

Motivation

A generic approach to detect attacks against confidentiality and integrity at different levels

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**



Different levels of DIFT

Coarse-grained approach: OS level

- Monitor system calls: containers = files, memory pages
- Pros & cons
  - \+ Monitor in kernel side protected from userland
  - \+ Tagging files is easier for the end user to specify its security policy
  - \+ Low runtime overhead
  - \- Over-approximation of application internal behavior
  - \- Cannot detect low-level attacks

Fine-grained approach: machine language level

- Monitor instruction execution: containers = registers, memory words
- Pros & cons
  - \+ Precise monitoring
  - \- Huge overhead and no isolation if implemented in software
  - \- Cannot tag persistent storage (files) if implemented in hardware

Originality of our approach

- **Combines hardware/software fine-grained DIFT with OS-level tagging** to associate labels to registers, memory and files
  - Helps the end-user to specify the security policy
  - Saves the security contexts between reboots
- Implements tag propagation in an **external co-processor** to isolate the monitor with **no modification of the main CPU**
- Main challenge: isolating the monitor in a dedicated co-processor creates a **semantic gap** between the monitor and the monitored system:
  - How can the isolated co-processor extract some information from the main CPU to infer the behavior of the monitored code?
- Solve the semantic-gap issue by an original combination of approaches:
  - pre-computing of **annotations** during the compilation of applications
  - sending of branching information using **hardware trace mechanisms**
  - sending of addresses of read/write accesses using **instrumentation** of the application code

Threat model

- We target software attacks that directly modify the values of containers (files, registers, memory)
- We do no handle physical attacks (*e.g.* fault injection using laser or physical side channel attacks)
- We only monitor applications
  - The OS kernel is part of our TCB
  - We could reduce the TCB to the kernel code that manages file tags and communicates with the co-processor

Use case and technological choices

Use case

- Embedded systems using rich OS in security critical contexts
  - Such systems cannot be redeveloped from scratch for economical reasons
  - Security concerns allow important modifications of existing systems if some level of compatibility with applications and drivers is achieved

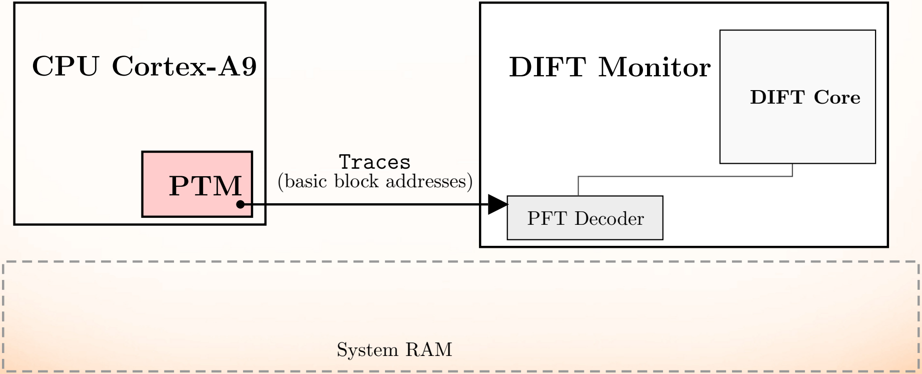
Software technological choices

- **Linux** embedded systems compiled with **LLVM** using **Yocto**
  - Open-source: implementation and evaluation of our approach
  - Very popular in embedded systems and simpler than Android

Hardware technological choices

- Digilent ZedBoard using Xilinx ZYNQ SoC
- Combine two hardcores (ARM Cortex A9) with an FPGA

PTM Traces


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

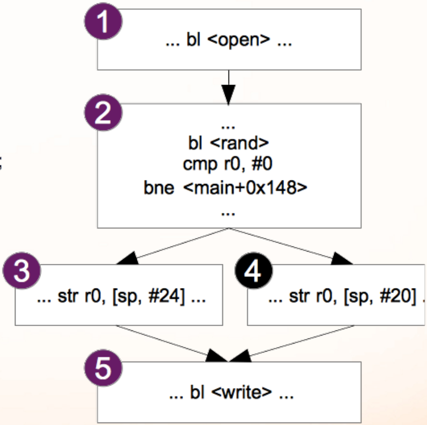
PTM Traces

CominLabs

```
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

CominLabs

Problem

We need to know what's happened between two jumps

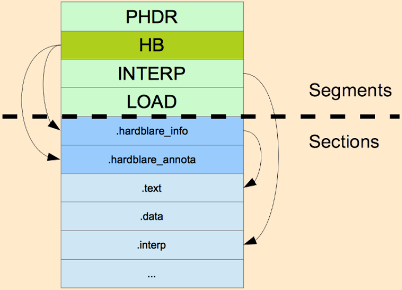
Solution

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

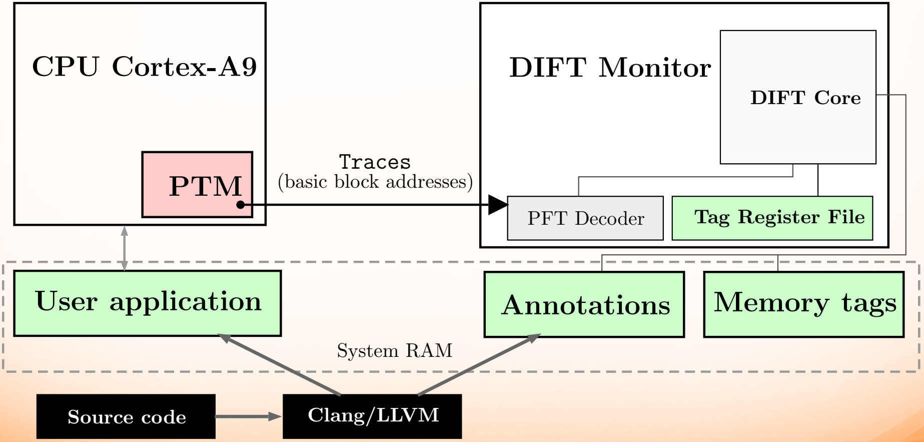
Examples :

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

Problem

Some addresses are resolved/calculated at run-time

Solution

- **Instrument the code** 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: different strategies

Recover **memory addresses**

Instruction	**Annotation**
ldr r1, [r2, #4]	$\underline{r1} \leftarrow \underline{\text{ mem } (r2+4)}$

Two possible strategies

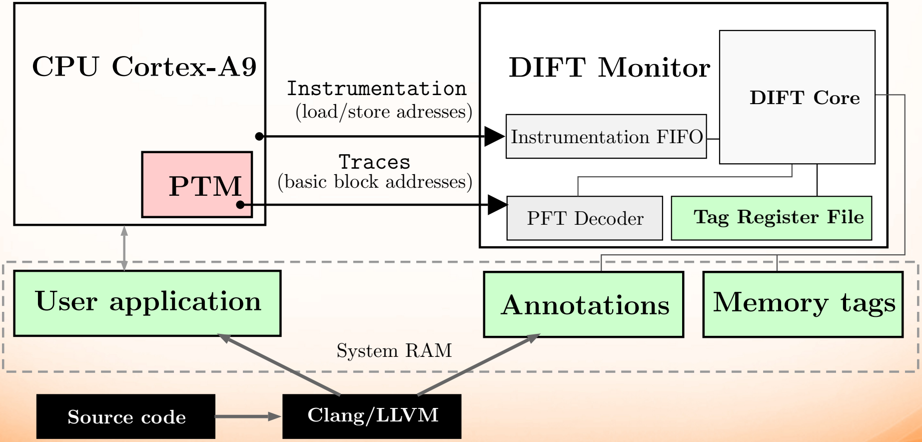
1. **Strategy 1:** Recover all memory address through instrumentation
2. **Strategy 2:** Recover only register-relative memory address through instrumentation

Instrumentation strategy 2

Recover only register-relative memory address through instrumentation

Example Instructions	Annotations	Memory address recovery
`sub r0, r1, r2`	r0 = r1 + r2	
`mov r3, r0`	r3 = r0	
`str r1, [PC, #4]`	@Mem(PC+4) = r1	CoreSight PTM
`ldr r3, [SP, #-8]`	r3 = @Mem(SP-8)	Static analysis
`str r1, [r3, r2]`	@Mem(r3+r2) = r1	instrumented

Instrumentation



RfBLare: handling system calls

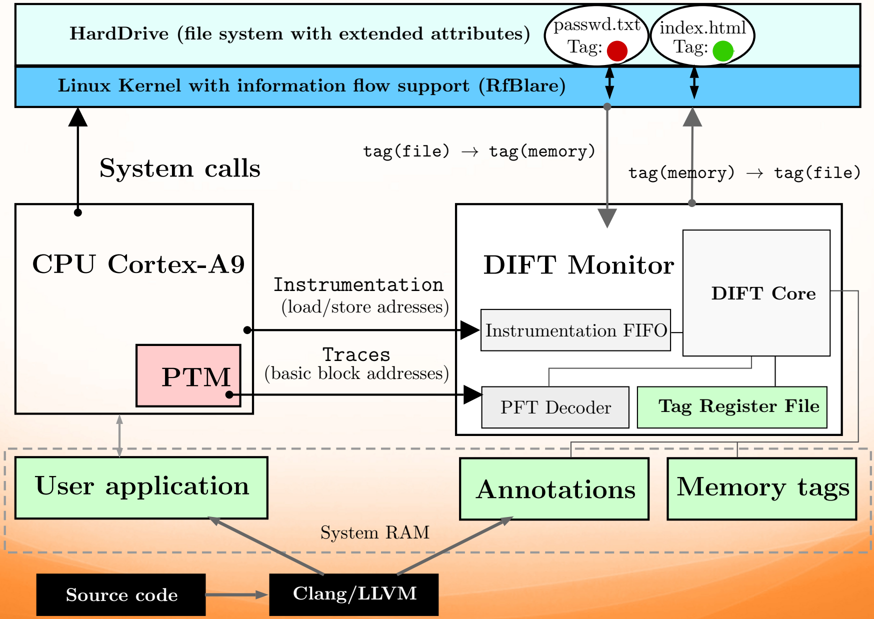
Problems

- We want to transmit tags from/to the operating system
- We want to persistently store tags in the system

Solutions

- Intercept syscalls using **Linux Security Modules Hooks**
- Attach labels to files in **Extended file attributes**
- The OS communicates with the co-processor to propagate tags:
  - When **reading** data from a file: $tag(file) \rightarrow tag(buffer)$
  - When **writing** data to a file: $tag(buffer) \rightarrow tag(file)$

RfBLare: System calls



Software developments

Software

- Modification of the Linux kernel:
  - LSM module to handle file tags
  - Communication with the co-processor
- Patch of the official Linux kernel PTM driver
  - Initial support of the ARM PTM trace mechanism was incomplete
  - The patch has been accepted by kernel maintainers [a]
- Modification of the Linux loader (`ld.so`) to load annotations
- Development of a LLVM backend pass
  - Compute annotations and save them in the elf binary file
  - Instrument application code to send read/write addresses
- All the software code is available on private project git repo
  - Access can be granted on demand
  - Will be published on public repo after the integration process

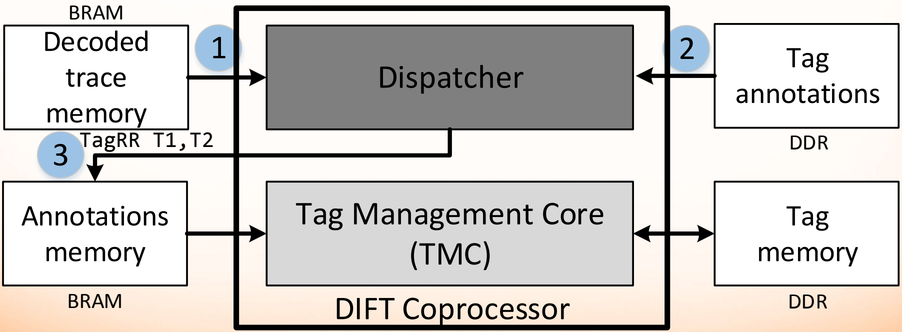
---

[a] `https://lore.kernel.org/patchwork/patch/723740/`

DIFT coprocessor[1]

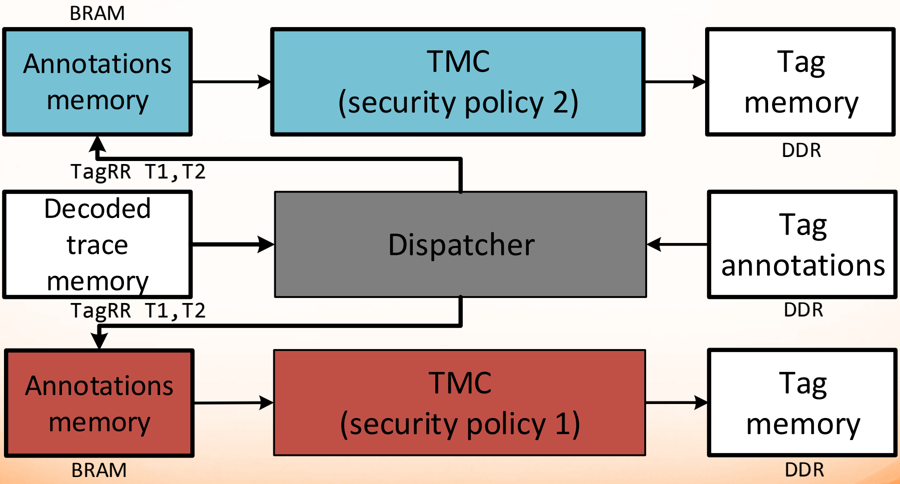
Two cores

- Dispatcher
- TMC (Tag Management Core)



[1] **reconfig_18**.

Use cases: Multiple security policies



Conclusion

Contributions

- Recovery of required information for DIFT on hardcore CPU
- Dedicated DIFT coprocessor for the ARM architecture
- Integration of OS support in the hardware-assisted DIFT
- Implementation of the proposed approach on the Zynq SoC
- Scalable solution for multiple security policies and multicore/multiprocessor systems

Perspectives

- Finalizing hardware integration and security evaluation
- Reducing the TCB, implementing isolation of kernel parts using TrustZone
- Reducing instrumentation overhead (by optimizing the static analysis)