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

Guillaume Hiet[1], Pascal Cotret[2], Vianney Lapotre[3], and Guy Gogniat[3]

[1]CIDRE, CentraleSupélec/Inria/CNRS, IRISA
[2]MOCS, ENSTA Bretagne, Lab-STICC
[3]MOCS, Université de Bretagne-Sud, Lab-STICC

Abstract

HardBlare proposes a hardware/software solution to implement an anomaly-based approach to detect software attacks against confidentiality and integrity through Dynamic Information Flow Tracking (DIFT). HardBlare combines fine-grained DIFT with OS-level tagging which allows end-user to specify security policy. Main outcome at the hardware level of HardBlare is the design of a dedicated multi-core DIFT co-processor on FPGA that does not require any modification of the main CPU. This contribution tackles one of the main challenges of the project. To achieve this goal, information required for DIFT is obtained both at compile-time and run-time through pre-computation during the compilation step, hardware trace mechanisms and instrumentation at run-time. Main outcome at the software level of HardBlare is the Linux kernel modification to handle file tags and to support communication between the instrumented code and the co-processor, a patch of the official Linux PTM driver, Linux loader (`ld.so`) modification to load annotations to the co-processor and a LLVM backend pass to compute the annotations and instrument applications. All validation has been performed on the Digilent ZedBoard using Xilinx ZYNQ SoC which combines two hardcores (ARM Cortex-A9) with a Xilinx FPGA. Through its achievements HardBlare demonstrates that a hardware/software co-design approach for Information Flow Control corresponds to an efficient and promising solution.

1 Context

Software security is still one of the main concerns in today's systems even though an important amount of research has been done on the subject. Indeed, systems are still affected by a high number of vulnerabilities and increasingly targeted by a wide range of sophisticated attacks. These attacks are strongly connected to underground economy and military/intelligence activities. They can combine different malicious behaviors, exploiting vulnerabilities at different levels, *e.g.* hijacking the control flow of a program by exploiting a buffer overflow or accessing some restricted part of the file-system by exploiting a directory traversal.

1.1 Embedded systems increasingly targeted by attackers

Recent attacks target embedded systems such as IoT devices, Industrial Control Systems and Cyber-Physical Systems in general. Security in embedded systems has often been neglected, resulting in a huge number of devices running with unpatched vulnerabilities. The goal of HardBlare is to enhance the security of embedded devices. However, embedded systems correspond to an important and heterogeneous set of systems. We can typically distinguish two different types of embedded systems:

- Systems using a RTOS (*Real Time Operating System*) and microcontrollers (*e.g.* ARM Cortex-M family), such as Programmable Logic Controllers or GPS receivers;
- **Systems using rich OS** (Linux, Android, etc.) and more powerful **application processors** (*e.g.* ARM Cortex-A family).

HardBlare targets this last type of systems, *e.g.* smartphones/tablets, smart watches, set-top boxes, business printers or military devices (*e.g.* Android Tactical Assault Kit [1]).

[1] `https://afresearchlab.com/technology/information-technology/tactical-assault-kit-tak/`

1.2 How embedded systems can be protected?

The best strategy to secure embedded systems would be to avoid vulnerabilities. Indeed many preventive approaches have been proposed, such as static analysis of software code, dynamic verification enforced by the runtime environment (*e.g.* the Android Virtual Machine) or use of cryprographic mechanisms. In practice, however:

- Preventive approaches are not systematically used (*e.g.* a lot of Android applications are still using C code and are not protected by the Android Virtual Machine runtime verification);
- These approaches are not sufficient to prevent all the attacks (*e.g.* using Java or OCaml does not prevent all the logical errors that could lead to potential vulnerabilities).

It is thus also important to **monitor** systems to **detect intrusions at runtime**. Detecting attacks or intrusions is just the first step of a reactive security and alerts can be used to notify security incidents to administrators, stop or modify the execution of system under attacks, put the system in quarantine, etc.

1.3 How to detect intrusions in embedded systems?

To detect software attacks against confidentiality and integrity at different levels (*e.g.* low-level attacks such as control-flow hijacking and more high-level attacks such as leaking of confidential files) we propose a generic anomaly-based approach. In that context, Dynamic Information Flow Tracking (DIFT) is a perfect candidate.

DIFT, illustrated by Figure 1.1, consists in:

1. Attaching **labels** called tags to **containers** (*e.g.* files, program variable or registers) and specifying an information flow **policy**, *i.e.* relations between tags;
2. **Propagating** tags at runtime to reflect information flows that occur during execution and **detecting** any **policy violation**.

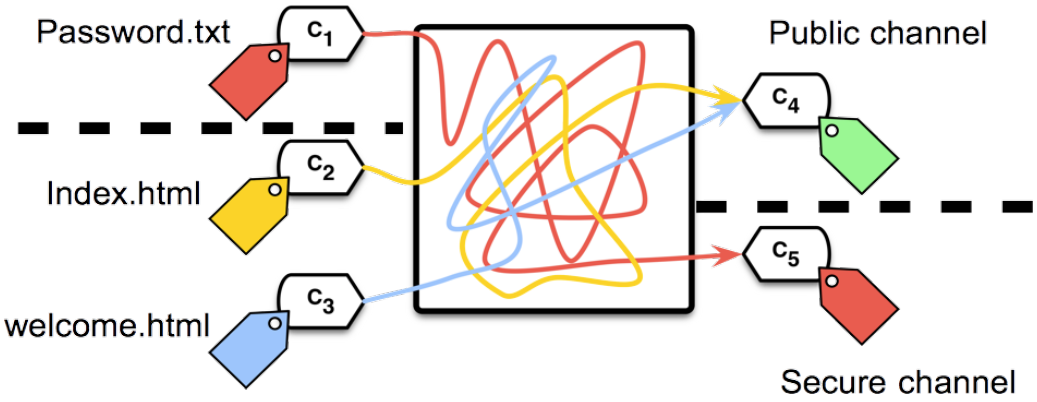

Figure 1.1: DIFT principle.

DIFT can be implemented at different levels. OS level DIFT [7, 6] is a coarse-grained approach which only monitors system calls. OS-level containers are files or memory pages. In such approaches, the monitor is usually implemented in kernel and is thus protected from userland applications. Specifying the policy consists in tagging files, which is quite easy for the end user. These approaches have also a relative low runtime overhead on the execution. However, they suffer from two major limitations:

- They over-approximate the internal behavior of applications. Thus they always consider that a read syscall on file f_1 followed by a write sycall on file f_2 performed by the same application results in an information flow from f_1 into f_2, even if the code executed by the application does not use the information from f_1 to compute the information written into f_2.

- They cannot be used to detect low-level attacks such as control-flow hijacking, which requires to handle more fine-grained containers such as registers.

Fine-grained approaches [4, 3] monitor each instruction executed by an application. They consider fine-grained containers such as registers or words stored into memory. Such approaches offer a more precise monitoring and can detect low-level attacks. However, they suffer from two major limitations:

- DIFT monitors implemented in software [5] are not isolated from their target (*i.e.* the monitor is weaved into the application code);
- They cannot tag persistent storage (files) if implemented in hardware [3] (*i.e.* the CPU is modified to store and propagate tags). Moreover, such hardware approaches are limited to softcore CPUs.

2 Approach developed in HardBlare

2.1 Originality of the approach

The originality of our approach lies in the following:

- We **combine fine-grained DIFT with OS-level tagging** to be able to attach tags to files. This helps the end-user to specify the security policy and can be used to save the security contexts between reboots (files being persistent containers).
- We implement tag propagation in a hardware co-processor to limit the overhead and isolate the monitor. Contrary to other hardware approaches, our co-processor approach requires **no modification of the main CPU**.
- Isolating the monitor in a dedicated co-processor creates a semantic gap between the monitor and the monitored system, *i.e.* how can the isolated co-processor extract some information from the main CPU to infer the behavior of the monitored code? Reducing this gap without modifying the main CPU is one of the main challenge of the project. We solve the semantic gap issue by an original combination of approaches:
  - We pre-compute **annotations** during the compilation of applications. Those annotations reflects the information flows in each basic block;
  - We send branching information using **hardware trace mechanisms** at runtime;
  - We send addresses of read/write accesses performed by the application using **instrumentation** of the application code.

2.2 Threat model

We consider the following threat model:

- We target software attacks that directly modify the values of containers (files, registers, memory);
- We do not 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 Trusted Computing Base (TCB). We could reduce the TCB to the kernel code that manages file tags and communicates with the co-processor.

2.3 Use case and technological choices

We target 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.

Our approach can be implemented on systems using a SoC FPGA that combines an ASIC CPU (PS) and a FPGA (PL) such as Xilinx ZYNQ SoC [2] or Intel/Altera Cyclone V SoC [3]. The monitored system is executed on

[2] `https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html`

[3] `https://www.intel.com/content/www/us/en/products/programmable/soc/cyclone-v.html`

the PS and the DIFT monitor is executed on the PL. In our experiment, we rely on the Digilent ZedBoard [4] using Xilinx ZYNQ SoC. This SoC combines two hardcores (ARM Cortex-A9) with a Xilinx FPGA.

The monitored system is a Linux-based system executed on the PS of the SoC. Linux-based systems are very popular in embedded systems and such open-source systems can be easily modified. We chose a traditional GNU/Linux system to implement our proof of concept since it is simpler to adapt than Android systems. However, our approach only needs some engineering effort to be ported to Android systems. We use tools from the Yocto project [5] to create and maintain our custom Linux distribution and LLVM [6] to implement static analysis and instrumentation of applications.

2.4 General Overview

Figure 2.1 illustrates the general overview of the HardBlare approach.

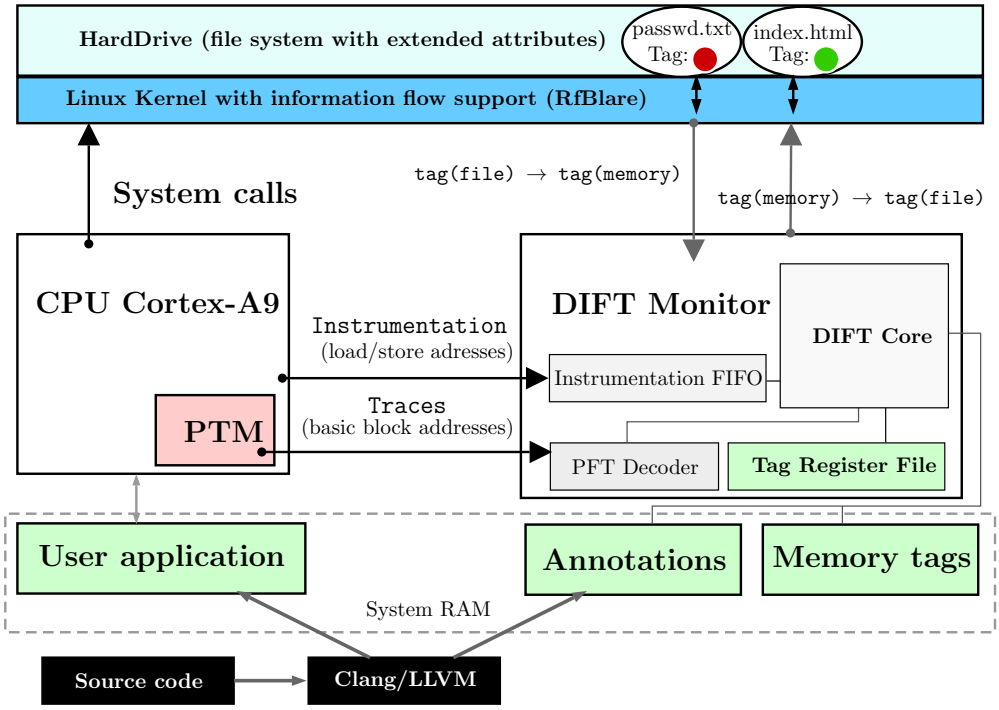

Figure 2.1: General overview of the HardBlare approach.

The DIFT monitor is implemented on the FPGA of the SoC (PL). It monitors the behavior of applications executed on the main CPU (Cortex-A9). It handles tags corresponding to fine-grained containers (CPU registers and words stored into the memory of the application). Those tags are themselves stored in two different memory regions:

- tags corresponding to registers of the main CPU are stored in *shadow registers* inside the FPGA (Tag Register File);
- tags corresponding to the application memory are stored into a dedicated part of the system DDR (Memory tags). This area can only be accessed by the DIFT Monitor and the OS nor the applications can access it.

To capture the runtime behavior of the applications and solve the semantic gap issue without modifying the main CPU, we rely on four communication channels: trace, annotation, instrumentation, and OS/Monitor channels.

We configure the ARM CoreSight PTM hardware trace mechanism so that the CPU automatically sends branching information to the monitor executed on the FPGA. This trace mechanism was originally designed

[4] http://zedboard.org/
[5] https://www.yoctoproject.org/
[6] https://llvm.org/

for debugging purpose but can be used to monitor the executed application in real-time for security purpose. As illustrated by our experimental results, the impact of this trace mechanism on the runtime performance is negligible. However, PTM can only send sparse information about the behavior of the monitored application. It only provides the address taken in branching instruction and events (such as exceptions). Thanks to this mechanism, the monitor can track the address of each basic block executed by the application. This information is however not sufficient to retrieve the information flow that occurred in each basic block.

During the compilation of each application of our Linux distribution, we statically analyze the code of the application and compute annotations for each basic block. Those annotations correspond to information flows that will occur in each basic block. They describe the tags propagation that the monitor will have to perform at runtime. We developed a dedicated LLVM pass to perform this static analysis. Those annotations are saved in a dedicated section of the elf binary file. We also modified the Linux loader (`ld.so`) to send the annotations to the co-processor when the application is loaded on the main CPU. At runtime, annotations are saved in a dedicated part of the DDR which is only accessible by the loader and the DIFT monitor. When a basic block of the application is executed on the main CPU, the PTM trace mechanism sends the address of this basic block to the DIFT Monitor. The monitor then retrieves the annotations corresponding to this basic block and executes them to propagate the tags stored inside Tag Register File and/or the tag memory. It also checks that the security policy is not violated and sends an interrupt to the OS in case of intrusion. The handling of this interrupt, *i.e.* the reaction part, is not in the scope of this project.

To handle tags corresponding to application memory, the monitor needs to know the virtual address of memory access to maintain a map between those addresses and the corresponding tags. Sometimes, the address of a memory access cannot be determined at compile time. In this case, the address is computed at runtime and saved in a register. To handle those cases, we instrument the program code so that it sends the address to the co-processor via a dedicated FIFO, just before accessing the memory. This instrumentation is implemented in a dedicated LLVM pass.

Finally, we associate a tag to each file of the system. Those tags are saved in the extended attributes of the file. Thus, when the application reads or saves some information from or to a file, we propagate the tag from the file to the memory buffer or vice-versa. Tags associated to files cannot be handled directly by the DIFT monitor but only by the OS. To handle such cases, we modified the Linux kernel. For each `read` syscall, our modified kernel sends the file tag to the co-processor thanks to a dedicated FIFO. For each `write` syscall, it retrieves the tag corresponding to the buffer from the FIFO and propagates it to the file attributes.

3 Project data and status

General information:

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

Partners

- IETR/CentraleSupélec (SCEE) @ Rennes
  - Pascal Cotret (Ass. Prof.) now at ENSTA Bretagne
  - Muhammad Abdul Wahab (PhD student) now R&D engineer at Ultraflux
- 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) now Research Fellow at NUS School of Computing, Singapore

4 Results

4.1 Hardware and software developments

Some developments have been pushed to public repositories:

- Implementation of the static instrumentation approach described in our AsianHOST'18 paper: `https://bitbucket.org/hardblare/hardware-assisted-static-instrumentation`
- Hardware and software related to the whole platform: `https://bitbucket.org/hardblare/hardblare-codes`

4.1.1 Hardware developments

We designed and implemented a dedicated multi-core DIFT co-processor on FPGA. This work has mainly been done in the context of the PhD of Muhammad Abdul Wahab. The architecture of this co-processor is illustrated by Figures 4.2 and 4.1. The first core (Dispatcher) is a generic fully-pipelined MIPS processor. It executes a firmware which parses the trace coming from the PTM, identifies the current basic block and retrieves the corresponding annotations from the annotations memory. These annotations are sent to the internal program memory of the second core, *i.e.* the Tag Management Core. The TMC executes the annotation to propagate tags and checks the security policy. Arnab Kumar Biswas developed a Tag Management Unit (TMU) during his PostDoc. This TMU is used by the TMC to retrieve the tag corresponding to a given virtual address.

In such multi-core design, we can use multiple TMCs. This can be useful to handle the following use-cases:

- Different tag propagation and check policies can be enforced in parallel (one policy by TMC). This is useful to detect different types of attacks.
- Different applications can be monitored in parallel (one application by TMC). This approach is useful to handle the difference in frequency between the main CPU and the DIFT monitor.

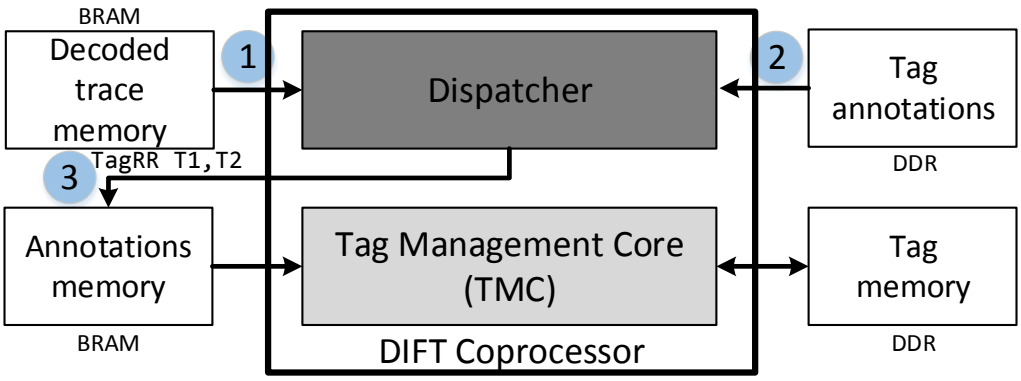

Figure 4.1: DIFT co-processor.

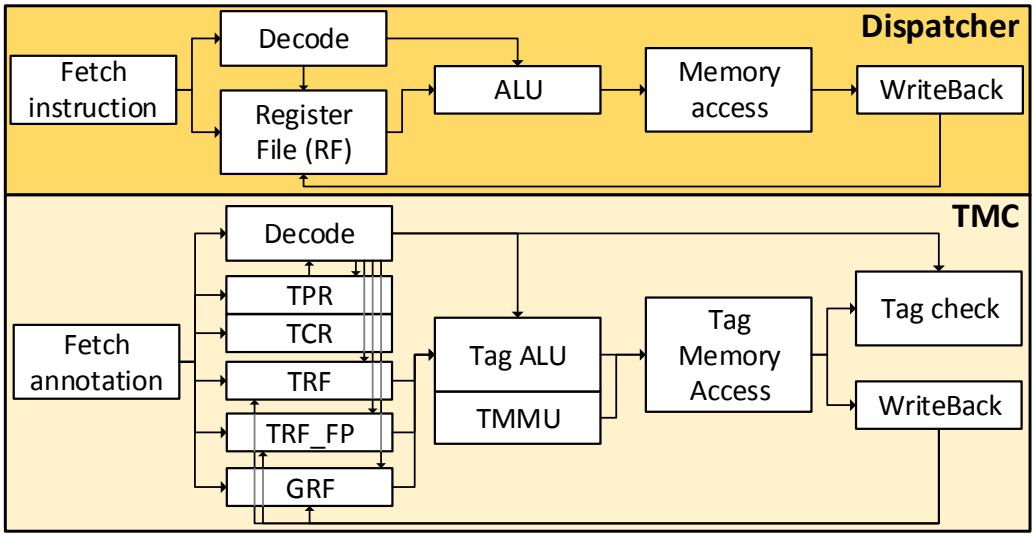

Figure 4.2: DIFT co-processor internal architecture.

4.1.2 Software developments

The software part of our approach is implemented into three distinct software components:

- We modified the Linux kernel to handle file tags and support communication between the instrumented code and the co-processor;
- We had to patch the official Linux PTM driver, which did not fully support all the features of the PTM. This patch has been officially adopted by the official vanilla Linux kernel distribution [7].
- We modified the Linux loader (`ld.so`) to load annotations to the co-processor;
- We developed a LLVM backend pass to compute the annotations and instrument applications.

This work has mainly been done in the context of the PhD of Mounir Nasr Allah.

4.2 Experimental results

Our experimental results are illustrated in Table 4.1. Compared to approaches from related work, HardBlare does not modify any part of the monitored CPU and can be used to monitor hardcore, *i.e.* ASIC CPU. This type of CPU is far more powerful than softcore synthesized on FPGA. The area overhead of the DIFT co-processor is limited, which allows to used multiple TMC to follow different applications in parallel. We are also able to monitor the code in the libraries used by the application and we support applications executed

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

on a rich OS (most of the existing approaches only support applications directly executed on the CPU). However our runtime overhead is still important and is mainly due to the communication overhead related to the instrumentation. For the moment, the instrumentation sends the address of every memory access which generates a lot of messages. We could reduce this traffic by optimizing the static analysis to better statically predict the address value.

These evaluations have been done using simulations on a simplified environment. We are currently working on the final integration of the last version of the hardware design with all the software stack. Once this integration will be achieved, we will conduct some security evaluations. We have prepared an experimental setup and we plan to evaluate the detection on two use-cases:

- a scenario where the attacker performed a directory traversal attack against the NGINX web-server [8] to access some confidential data (*e.g.* the system password file);
- a scenario where the attacker exploits different vulnerabilities in simple code examples illustrating the common weaknesses of C program, listed in Common Weakness Enumeration [9].

We plan to conduct such experiments within six months.

	Without OS support			**with OS support**	
Approaches	**Kannan [3]**	**Deng [1]**	**Heo [2]**	**Heo** [2] **adapted**	**HardBlare**
Area overhead	6.4%	14.8%	14.47%	N/A	**0.95%**
Power overhead	N/A	**6.3%**	24%	N/A	16.2%
Max frequency	N/A	**256 MHz**	N/A	N/A	250 MHz
Communication time overhead	N/A	N/A	60%	1280%	**335%**
Hardcore portability	No	No	**Yes**	**Yes**	**Yes**
Main CPU	Softcore	Softcore	Softcore	**Hardcore**	**Hardcore**
Library instrumentation	N/A	N/A	partial	**Yes**	**Yes**
FP support	No	No	No	No	**Yes**
Multi-threaded support	No	No	No	No	**Yes**

Table 4.1: Experimental results and comparison with state-of-the-art.

4.3 Internal collaborations

- Strong collaboration between partners to propose a global hardware/software approach and develop a full prototype.
- PhD students have been co-advised by permanent researchers from the different institutions.
- Muhammad Abdul Wahab defended his PhD in December 2018. The PhD Defense of Mounir Nasr Allah is planned in June 2020.

4.4 External collaborations and dissemination

- Mounir Nasr Allah did an internship of six months at ARM Cambridge with Alastair Reid (07/2017 to 01/2018):

[8] `https://www.nginx.com/`
[9] `https://cwe.mitre.org/data/definitions/658.html`

  - Working on model checking of the formal specification of ARM Cortex M processors to verify IFC properties;
  - Opportunity to present our work to ARM and to better understand the ARM processor architecture.

- Muhammad Abdul Wahab did a three months internship at ALaRI Lugano with Alberto Ferrante (01/2018 to 03/2018):
  - Exploring how trace mechanisms and FPGA of the ZYNQ SoC can be used to accelerate malware detection.

- We have been contacted by Norwegian researchers from HVL, which are interested by our approach. We submitted a project proposal with them to the European AURORA collaboration project call for proposal.

- We have presented our work on this project to the following industrial partners : ARM research (Cambridge, UK), HP Labs (Bristol, UK), Secure-IC (Rennes, France), IBM OpenPower team (Rochester, USA).

4.5 Publications

- **International Conferences with proceedings (3 + 1 short paper)**
  - Abdul Wahab et al.: *A small and adaptive coprocessor for information flow tracking in ARM SoCs*, ReConFig2018
  - Abdul Wahab et al.: *A novel lightweight hardware-assisted static instrumentation approach for ARM SoC using debug components*, AsianHOST2018
  - Abdul Wahab et al.: *ARMHEx: A hardware extension for DIFT on ARM-based SoCs*, FPL2017
  - Abdul Wahab et al.: *Towards a hardware-assisted information flow tracking ecosystem for ARM processors (short paper)*, FPL2016

- **International technical conferences (3)**
  - HITBSecConf 2017, 34th Chaos Communication Congress 2017, Toulouse Hacking Convention 2018

- **National conferences and workshops (3)**
  - CryptArchi 2016, 11ème Colloque National du GDR SoC/SiP, RESSI2017

- **Invited talks (4)**
  - P. Cotret. *Towards a hardware-assisted information flow Tracking Approach for ARM Processors*. France/Japan Cybersecurity workshop, France, 2016.
  - G. Gogniat. *ARMHEx: A hardware extension for DIFT on ARM-based SoCs*, Ecole d'hiver Francophone sur les Technologies de Conception des Systèmes embarqués Hétérogènes (FETCH 2018), Saint-Malo, France, 2018.
  - G. Gogniat. *ARMHEx: A hardware extension for DIFT on ARM-based SoCs*, 16th International Conference on Frontiers of Information Technology (FIT 2018), Islamabad Pakistan, 2018.
  - V. Lapotre, *A Hardware/software co-design approach for security analysis of application behavior - Applications on Dynamic Information Flow Tracking* , Journée "Nouvelles Avancées en Sécurité des Systèmes d'Information", Toulouse, France, 2019.

- **Posters (2)**
  - CHES 2015, séminaire doctorants SIF 2016

5 Perspectives

We have identified five main perspectives:

- Reducing the TCB, implementing isolation of kernel parts using TrustZone;
- Reducing instrumentation overhead (by optimizing the static analysis);
- Implementing multicore and multi-thread DIFT (by using multiple TMC);
- Porting the approach to other platforms (*e.g.* Intel PT);
- Taking benefit of dynamic partial reconfiguration of FPGA to increase the flexibility of the co-processor.

References

[1] D. Y. Deng, D. Lo, G. Malysa, S. Schneider, and G. E. Suh. Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric. In *IEEE/ACM International Symposium on Microarchitecture.*, pages 137–148. IEEE Computer Society, 2010.

[2] I. Heo, M. Kim, Y. Lee, C. Choi, J. Lee, B. B. Kang, and Y. Paek. Implementing an application-specific instruction-set processor for system-level dynamic program analysis engines. *ACM TODAES.*, 20(4):53, 2015.

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

[4] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 85–96, New York, NY, USA, 2004. ACM.

[5] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Acm Sigplan Notices.*, volume 39, pages 85–96. ACM, 2004.

[6] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 19–19, Berkeley, CA, USA, 2006. USENIX Association.

[7] J. Zimmermann, L. Mé, and C. Bidan. *Introducing Reference Flow Control for Detecting Intrusion Symptoms at the OS Level*, pages 292–306. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.