ARMHEx: embedded security through hardware-enhanced information flow tracking

Muhammad Abdul – Wahab, Pascal Cotret, Mounir – Nasr Allah, Guillaume – Hiet, Vianney Lapotre, Guy Gogniat

ARMHEx: embedded security through hardware-enhanced information flow tracking

Muhammad A. Wahab*, Pascal Cotret*, Mounir N. Allah†, Guillaume Hiet†, Vianney Lapôtre‡ and Guy Gogniat‡
*IETR, SCEE – †INRIA, CIDRE – CentraleSupélec, Rennes, FRANCE firstname.lastname@centralesupelec.fr
‡Lab-STICC laboratory – University of South Brittany, Lorient, FRANCE firstname.lastname@univ-ubs.fr

***Abstract*—Security in embedded systems is a major concern for several years. Untrustworthy authorities use a wide range of both hardware and software attacks. This paper introduces ARMHEx, a practical solution targeting DIFT (*Dynamic Information Flow Tracking*) implementations on ARM-based SoCs. DIFT is a solution that consists in tracking the dissemination of data inside the system and permit to ensure some security properties. Existing DIFT solutions are either hardly portable to SoCs or bring unsuitable time overheads. ARMHEx overcomes both issues using modern debugging CPU features, along with a coprocessor implemented in FPGA logic. This work demonstrates how ARMHEx performs DIFT with negligible communication costs opening interesting perspectives in the context of reconfigurability and hardware-enhanced security for multiprocessor architectures.**

I. Introduction

During the last decade, several security vulnerabilities have been discovered. Even if patches were delivered, there is always a game of cat and mouse between security developers and hackers. Embedded systems are a target of choice for attackers. Indeed many vulnerabilities have been discovered on such systems. In the meantime, those systems often contain confidential data and services which require a high level of integrity. A first solution to tackle this problem consists in reducing the number of vulnerabilities by using different techniques such as patch management, careful code reviews, static analyses or by choosing managed languages (e.g. Java) that are considered more robust. However, none of these techniques are sufficient, in practice, to ensure the absence of vulnerabilities on a complex system made of multiple applications.

Access control or cryptography can be used to restrict accesses to confidential data or to enforce integrity. However, they do not provide any guarantees once access is granted or data decrypted. Monitoring applications at runtime to check their behavior is a complementary solution. Among the different existing approaches, IFT (*Information Flow Tracking*) is an appealing solution that consists in tracking the dissemination of data inside the system. Two approaches can be defined:

- SIFT (*Static Information Flow Tracking*). This is an offline analysis of the application aiming to check that all branches of the control flow graph are trustworthy.
- DIFT (*Dynamic Information Flow Tracking*). DIFT is performed at runtime: it monitors the application binary in order to check if the execution is safe.

This work is based on an hybrid approach combining SIFT and DIFT [1]: both dynamic and hybrid IFTs will be cited as DIFT or IFT in the following. DIFT consists of performing three operations:

1) **Tag initialization**: Each information container (e.g. file, variable, memory word, etc) is given a tag. Those tags corresponds to the security level or the type of data they contained.
2) **Tag propagation**: Each time an instruction is executed on the CPU, tags are propagated from source operands to destination operands to track information flows.
3) **Tag Check**: To ensure that critical information is not handled by untrusted functions or entities, tags are checked with a security policy at runtime and on a regular basis.

This paper is organized as follows. Section II introduces main contributions regarding DIFT solutions. Our proposed solution ARMHEx is described in Section III. Then, implementation results are given in Section IV and compared to state of the art work. Finally, Section V gives some conclusions and future perspectives for ARMHEx.

II. Related work

Software solutions for IFT are generally unusable in practice. For single-core architectures, the CPU must execute the main application and IFT-related operations. Therefore, extensive time overheads (at least 300%) can be expected as the same hardware unit has to perform both operations [2], [3], [4]. To overcome those overheads, hardware mechanisms were implemented in DIFT solutions. We can distinguish four main approaches:

1) **Filtering hardware accelerator** [5], [6]. Instead of computing tags for each CPU instruction (as done in other approaches), this approach proposes to filter monitored events (e.g. instructions or system calls) before computing tags to lower DIFT time overhead.
2) **In-core** [7], [8]. This solution relies on a deeply revised processor pipeline. Each stage of the pipeline is duplicated with a hardware module in order to propagate tags all along the program execution.
3) **Offloading** [9], [10]. In this case, DIFT operations are computed by a second general purpose processor.
4) **Off-core** [11], [12], [13], [14], [15], [16]. This solution seems similar to the offloading one. However, DIFT is performed on a dedicated unit instead of a general purpose processor. ARMHEx is based on this approach but differs in its implementation: the application runs on a hardcore (rather than softcore as in previous works) and the information required for DIFT is recovered through debug components and modified compiler.

Table I compares features in existing works that are based on the same off-core approach as ARMHEx. [11], [12], [13] implemented DIFT using a softcore processor. In these works, there are modifications of the CPU itself in order to export information needed for DIFT. The required information is recovered from existing CPU signals which makes this approach not portable on hardcore. The other related works mentioned in Table I are hardcore portable but present time overhead due to the communication interface used between the CPU and the DIFT coprocessor.

TABLE I: Features comparison with related work (Off-core approaches)

Related work	Experimental Target	Communication interface	Hardcore portability
Kannan et al. [11] Deng et al. [12],[13]	Softcore	Signals and FIFO	No
Heo et al. [14] Davi et al. [15]	Softcore	Binary instrumentation	Yes
Lee et al. [16]	Softcore	CDI(*Core Debug Interface*), Caches	Yes
ARMHEx	Zynq SoC (ARM + FPGA)	CoreSight components, Compiler customized	Yes

Lee et al. [16] work appears closest to ARMHEx. However, there are lot of implementation details that differ. The striking difference is that the debug interface considered in [16] is ARM ETM CoreSight component which can provide information for each CPU instruction. In ARMHEx, more recent ARM PTM CoreSight component is considered which can provide information only on some instructions that modify the PC register value. Furthermore, their proposed implementation prototype is based on Leon3 softcore. This simplifies the implementation as the real targeted device is not used for experiments and the problems related to implementation (e.g. missing features in driver) are not tackled. Besides, all the implementations done in related works target a softcore CPU which explains their lack of deployment in industry. ARMHEx framework proposes and implements an approach that is portable to hardcore CPUs and that takes profit of modern CPU features.

III. ARMHEx APPROACH

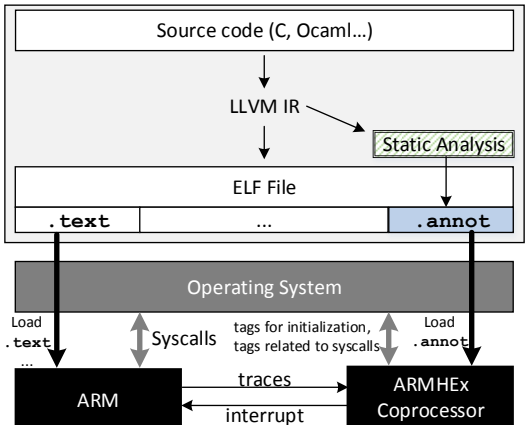

Fig. 1: Overall architecture

A. General Overview

Figure 1 sums up the overall architecture of both software and hardware parts. The source code file is compiled to obtain the executable ELF file. During compilation, static analysis is done to get an additional section `.annot`. This section contains tag propagation instructions that are executed by ARMHEx coprocessor. It is loaded by the OS to a memory accessible by ARMHEx coprocessor when binary (ELF file) is launched on ARM CPU. The operating system sends information on tag initialization operation and system calls to ARMHEx coprocessor. Traces are recovered by ARMHEx coprocessor thanks to CoreSight components.

In order to decouple application execution from tag computation, ARMHEx coprocessor requires at least three pieces of information to compute DIFT operations: (i) PC register value, (ii) instruction encoding and (iii) `load`/`store` memory addresses. By using CoreSight components, PC register value and some memory addresses are partially retrieved. Missing information about predictible memory addresses and instruction encoding is obtained through static analysis.

TABLE II: Example code and corresponding trace

Assembly code	Trace packets	Analysis
–	A-Sync	
–	I-Sync	
860c: sub sp, sp, #28	-	static
8610: bl 8480	BAP	dynamic
8614: mov r3, r0	-	static
8618: cmp r3, #0	-	static
861c: beq 864c	BAP/Atom	dynamic

B. ARMHEx software requirements

ARMHEx uses static analysis to recover partial information required for DIFT analysis. For instance, if the code presented in Table II is considered, the information about `sub`, `mov` and `cmp` instructions will be obtained through static analysis. As a result, a corresponding tag propagation instruction will be obtained for each of these instructions. Some examples of tag propagation instructions are shown in Table III. $\underline{R}$ is used to denote the tag of register `R`. For instance, for the first instruction in Table III, the corresponding propagation instruction is to associate tags of operand `R1` and `R2` towards the tag of destination register `R0`. A section `.annot` is added to the binary during compilation which contains all the tag propagation instructions that need to be executed by ARMHEx coprocessor. This extra section is ignored by the Linux kernel at runtime.

TABLE III: Example tag propagation instructions

Example Instruction	Corresponding tag propagation instruction
`ADD R0,R1,R2`	$\underline{R0} = \underline{R1}$ OR $\underline{R2}$
`LDR R3, [SP+OFFSET]`	$\underline{R3} = \underline{@Mem(SP+OFFSET)}$
`STR R0, [R5,R1]`	$\underline{@Mem(R5+R1)} = \underline{R0}$

In order to make the analysis easier, the application code is divided into basic blocks. ARMHEx considers that a basic block is a list of sequential instructions which ends with a branch instruction. For each basic block, all information flows between containers are listed using static analysis done using LLVM. This compiler is now quite popular and used by many vendors including Apple. It is very modular, which facilitates the implementation of static analysis. LLVM is composed of three main parts as shown in Figure 2.

1) The front-end in charge of reading the source code on a specific high-level language and converting it to LLVM-IR (*Intermediate Representation*).

2) The most common optimizer operations where all LLVM-IR optimizations are processed.
3) The back-end responsible of adapting the LLVM-IR into ARM instructions.

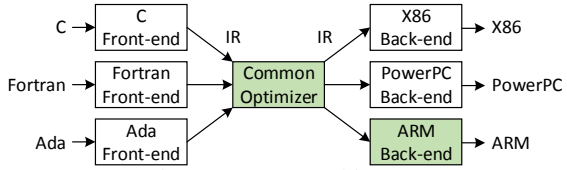

Fig. 2: LLVM architecture

For static analysis, ARMHEx needs the CFG (*control flow graph*), variable types and size regardless of the application programming language. Such information can be obtained from LLVM-IR and ARM back-end (colored blocks in Figure 2).

C. CoreSight components

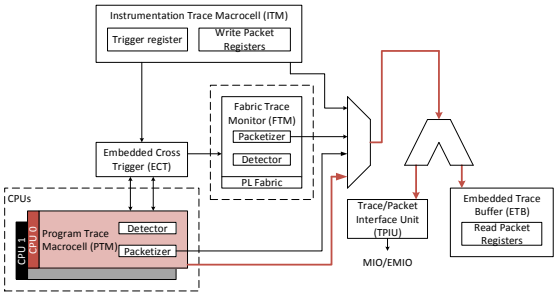

Fig. 3: CoreSight components in Xilinx Zynq

CoreSight components (Figure 3) are a set of IP blocks providing hardware-assisted software tracing. These components are used for debug and profiling purposes. For instance, they can be used to find software bugs and errors or even for CPU profiling (number of cache misses/hits). They are present in Cortex-A, Cortex-M and Cortex-R families of ARM processors. ARMHEx uses these components to retrieve information on instructions committed by the CPU: as a consequence, it can be done only at runtime. Table II shows that the trace always starts with synchronization packets `A-Sync` and `I-Sync`. Then `bl` and `beq` instructions generate trace packets. If a `BAP` packet is generated, the branch was taken. Otherwise, an `atom` packet is generated. The Linux driver for CoreSight components was not fully featured. We developed a patch that is under integration in the next Linux kernel release.

IV. Implementation results

Implementations were done on a Xilinx Zedboard including a Z-7020 SoC (dual-core Cortex-A9 running at 667MHz and an Artix-7 FPGA). Vivado 2014.4 tools were used for synthesis. The FPGA logic has around 85,000 logic cells and 560 KB of Block RAMs. Microblaze is used as DIFT coprocessor for a proof of concept.

TABLE IV: Performance comparison with off-core approaches

Approaches	Kannan [11]	Deng [12]	Heo [14], Lee [16]	ARMHEx
Hardcore portability	No	No	Yes	Yes
Communication overhead	high	high	high	negligible
Surface overhead	small	high	high	high
Main CPU	Softcore	Softcore	Softcore	Hardcore
Max frequency	N/A	256 MHz	N/A	250 MHz

Table IV shows a performance comparison of ARMHEx with previous off-core approaches. Unlike previous works, ARMHEx has the benefit of being based on an ARM hardcore processor: it opens interesting perspectives as this work is easily portable to existing embedded systems. Approaches proposed by Heo [14] and Lee [16] requires architectural modifications to be implemented on other SoCs. In addition, the time cost for communication between CPU and ARMHEx coprocessor is negligible thanks to CoreSight components. MiBench programs showed negligible runtime overhead. All other related works present non-negligible communication overhead. Furthermore, ARMHEx is able to operate at a frequency up to 250 MHz (bridled at 100 MHz for the first implementation because of a Microblaze used for DIFT computations). In terms of area, ARMHEx is not the most competitive solution: even if the area regarding the FPGA capacity is encouraging, this work is based on a MicroBlaze softcore for DIFT computations which is oversized for such an application.

V. Conclusion and perspectives

This work combines an offline static analysis and the use of CoreSight components to partially recover required information for DIFT with negligible time overhead. Further experimentations need to be done in order to evaluate proposed architecture and different security policies. As ARMHEx takes less than 22% of FPGA area, the second Cortex-A9 core can be protected as well by adding another DIFT coprocessor.

References

[1] S. Moore et al. Static analysis for efficient hybrid information-flow control. In *CSF 11*, June 2011.
[2] James Newsome et al. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.
[3] L. C. Lam et al. A general dynamic information flow tracking framework for security applications. 2006.
[4] Arnar Birgisson, Daniel Hedin, and Andrei Sabelfeld. *Boosting the Permissiveness of Dynamic Information-Flow Tracking by Testing*, pages 55–72. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
[5] Sotiria Fytraki et al. FADE: A programmable filtering accelerator for instruction-grain monitoring. In *HPCA 2014*, pages 108–119, 2014.
[6] Shimin Chen et al. Flexible hardware acceleration for instruction-grain lifeguards. *IEEE Micro*, 29, 2009.
[7] Michael Dalton et al. Raksha: A flexible information flow architecture for software security. *SIGARCH Comput. Archit. News*, June 2007.
[8] G. Venkataramani et al. Flexitaint: A programmable accelerator for dynamic taint propagation. In *HPCA 08*, pages 173–184, Feb 2008.
[9] Vijay Nagarajan et al. Dynamic information flow tracking on multicores. *Workshop on Interaction between Compilers and Computer Architectures*, 2008.
[10] Kangkook Jee et al. Shadowreplica: Efficient parallelization of dynamic data flow tracking. CCS '13, 2013.
[11] H. Kannan et al. Decoupling dynamic information flow tracking with a dedicated coprocessor. In *DSN 09*, pages 105–114, June 2009.
[12] Daniel Y. Deng et al. Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric. MICRO '43, 2010.
[13] Daniel Y. Deng et al. High-performance parallel accelerator for flexible and efficient run-time monitoring. DSN '12, pages 1–12, 2012.
[14] Ingoo Heo et al. Implementing an application-specific instruction-set processor for system-level dynamic program analysis engines. *ACM Trans. Des. Autom. Electron. Syst.*, 20(4):53:1–53:32, September 2015.
[15] L. Davi et al. Hafix: Hardware-assisted flow integrity extension. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2015.
[16] Jinyong Lee, Ingoo Heo, Yongje Lee, and Yunheung Paek. Efficient dynamic information flow tracking on a processor with core debug interface. DAC '15. ACM.