ARMHEx: a hardware extension for information flow tracking on ARM-based platforms

Muhammad Abdul Wahab[α], Pascal Cotret[α], Mounir Nasr Allah[β], Guillaume Hiet[β]
Vianney Lapôtre[γ], Guy Gogniat[γ]
[α] IETR / SCEE research group, firstname.lastname@centralesupelec.fr
[β] INRIA / CIDRE research group, firstname.lastname@centralesupelec.fr
[γ] Lab-STICC / University of South Brittany, 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. 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.*

1 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. 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. However, none of these techniques are sufficient, in practice, to ensure the absence of vulnerabilities on a complex system made of multiple applications. DIFT (*Information Flow Tracking*) is an appealing solution that consists in tracking the dissemination of data inside the system. 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 2 introduces main contributions regarding DIFT solutions. Our proposed solution ARMHEx is described in Section 3. Then, implementation results are given in Section 4 and compared to state of the art work. Finally, Section 5 gives some conclusions and future perspectives for ARMHEx.

2 Related work

Software solutions for DIFT are generally unusable in practice. For single-core architectures, the CPU must execute the main application and DIFT-related operations. Therefore, extensive time overheads (at least 300%) can be expected as the same hardware unit has to perform both operations [10, 7, 1]. To overcome those overheads, hardware mechanisms were implemented in DIFT solutions. We can distinguish three main approaches:

1. **In-core** [2]. This approach 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.
2. **Offloading** [9]. In this case, DIFT operations are computed by a second general purpose processor.
3. **Off-core** [6, 4, 5, 3]. This approach 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.

3 ARMHEx approach

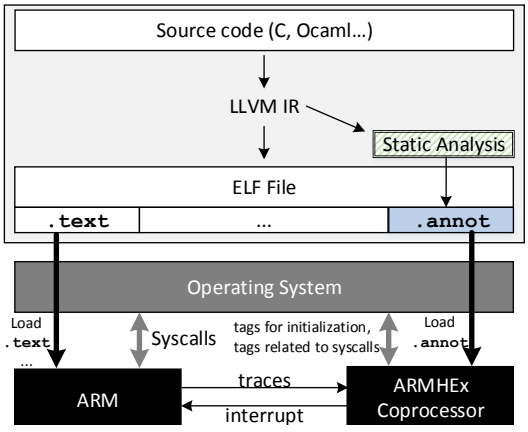

Figure 1: Overall architecture

3.1 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 memory addresses and instruction encoding is obtained through static analysis.

Table 1: 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

3.2 ARMHEx software requirements

ARMHEx uses static analysis to recover partial information required for DIFT analysis. For instance, if the code presented in Table 1 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 2. `R` is used to denote the tag of register `R`. For instance, for the first instruction in Table 2, the corresponding propagation instruction is to associate tags of operand `R1` and `R2` towards the tag of destination register `R0`. A section `.annot`, ignored by the Linux kernel, is added to the binary during compilation which contains all the tag propagation instructions that need to be executed by ARMHEx coprocessor.

Table 2: Example tag propagation instructions

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

3.3 CoreSight components

CoreSight components 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 1 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.

4 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 2016.4 tools were used for synthesis and implementation. The FPGA logic has around 85K logic cells and 560 KB of Block RAMs. Microblaze is used as DIFT coprocessor for a proof of concept.

Table 3 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 [5] and Lee [8] requires architectural modifications to be implemented on other SoCs. In terms of area, ARMHEx has the best processor/coprocessor ratio. This is because, ARMHEx targets ARM Cortex-A9 core which has a large number of gates compared to softcores considered in related work. In addition, the communication cost between CPU and ARMHEx coprocessor is 90% better than Heo et al. [5]. 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).

Table 3: Performance comparison with off-core approaches

Approaches	Kannan [6]	Deng [4]	Heo [5], Lee [8]	ARMHEx
Hardcore portability	No	No	Yes	Yes
Main CPU	Softcore	Softcore	Softcore	Hardcore
Communication overhead	N/A	N/A	60%	6.4%
Surface overhead	7.64%	14.8%	14.47%	<0.01%
Max frequency	N/A	256 MHz	N/A	250 MHz

5 Conclusion and perspectives

ARMHEx is the first work to implement DIFT on ARM hardcore processors. Even though DIFT implementations on softcores exist, they are not all portable to hardcore CPUs. This work proposes to use CoreSight components to partially recover required information for DIFT with negligible time overhead. ARMHEx extensions can be implemented in parallel as it has a moderate impact in terms of area as less than 22% of FPGA area is currently used. Implementation results show interesting perspectives for ARMHEx in terms of reconfigurability (flexible security policy changes) and multicore runtime security.

References

[1] A. Birgisson, D. Hedin, and A. Sabelfeld. *Boosting the Permissiveness of Dynamic Information-Flow Tracking by Testing*, pages 55–72. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[2] M. Dalton et al. Raksha: A flexible information flow architecture for software security. *SIGARCH Comput. Archit. News*, June 2007.

[3] L. Davi et al. Hafix: Hardware-assisted flow integrity extension. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2015.

[4] D. Y. Deng et al. Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric. MICRO '43, 2010.

[5] I. 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, Sept. 2015.

[6] H. Kannan et al. Decoupling dynamic information flow tracking with a dedicated coprocessor. In *DSN 09*, pages 105–114, June 2009.

[7] L. C. Lam et al. A general dynamic information flow tracking framework for security applications. 2006.

[8] J. Lee, I. Heo, Y. Lee, and Y. Paek. Efficient dynamic information flow tracking on a processor with core debug interface. DAC '15. ACM.

[9] V. Nagarajan et al. Dynamic information flow tracking on multicores. *Workshop on Interaction between Compilers and Computer Architectures*, 2008.

[10] J. Newsome et al. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.