ARMHEx: A hardware extension for DIFT on ARM-based SoCs

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

ARMHEx: A hardware extension for DIFT on ARM-based SoCs

Muhammad A. Wahab*, Pascal Cotret*, Mounir N. Allah†, Guillaume Hiet†, Vianney Lapôtre‡ and Guy Gogniat‡
*IETR, SCEE research group – CentraleSupélec, Rennes, FRANCE
Email: firstname.lastname@centralesupelec.fr
†INRIA, CIDRE research groupe – CentraleSupélec, Rennes, FRANCE
Email: firstname.lastname@centralesupelec.fr
‡Univ. Bretagne-Sud, UMR CNRS 6285, Lab-STICC, Lorient, France,
Email: firstname.lastname@univ-ubs.fr

***Abstract*—Security is a major issue nowadays for the embedded systems community. Untrustworthy authorities may use a wide range of attacks in order to retrieve critical information. This paper introduces ARMHEx, a practical solution targeting DIFT (*Dynamic Information Flow Tracking*) on ARM-based SoCs (e.g. Xilinx Zynq). Current DIFT implementations suffer from two major drawbacks. First, recovering required information for DIFT is generally based on software instrumentation leading to high time overheads. ARMHEx takes profit of ARM CoreSight debug components and static analysis to drastically reduce instrumentation time overhead (up to 90% compared to existing works). Then, security of the DIFT hardware extension itself is not considered in related works. In this work, we tackle this issue by proposing a solution based on ARM Trustzone.**

I. Introduction

During the last decade, several software security vulnerabilities have been discovered. Even if patches were delivered, there is always a game of cat and mouse between security developers and hackers on such systems containing confidential data and services which require a high level of integrity. A first solution to tackle this issue consists in using different techniques such as patch management, careful code reviews, static analyses or by choosing managed languages such as Java that are considered more robust. However, none of these techniques are sufficient in practice to get rid of all vulnerabilities.

Access control or cryptography can be used to limit access to confidential data or to enforce integrity. However, such techniques do not provide any guarantees once access is granted or data decrypted. Therefore, hardware manufacturers are more and more inclined to integrate specific security features in their CPUs: for instance, ARM has proposed Trustzone and Intel has recently proposed SGX. Such solutions are mainly used to isolate trusted software from other untrusted applications. They cannot protect untrusted software which is the goal of this work. 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. SIFT is mainly used for testing an application binary.
- DIFT (*Dynamic Information Flow Tracking*). DIFT is performed at runtime: it monitors data flow of the application binary in order to check if the execution is safe. DIFT is used to protect against software attacks (such as buffer overflows, format-string attacks, SQL injection, cross-site scripting, etc.) or used for data leakage prevention.

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

1) **Tag initialization**: it consists in attaching tags to information containers (e.g. file, variable, memory word, etc). Those tags corresponds to the security level or the type of data they contain.
2) **Tag propagation**: tags need to be propagated from source operands to destination operands to track information flows resulting from the execution of each CPU instruction.
3) **Tag Check**: tags are checked with a security policy, at runtime and on a regular basis, to ensure that critical information is not handled by untrusted functions or entities.

Solutions relying on software methods are penalized by high time overheads: even if system security is preserved, software DIFT is unusable in practice. Recently, an interest has arisen in heterogeneous architectures (including a hardcore processor and reconfigurable logic) for DIFT mechanisms: hardware modules are used to improve DIFT performance while keeping strong security properties. Furthermore, recent technologies combining hardcore processors with reconfigurable logic open interesting features in the context of hardware/software information flow tracking.

This paper is organized as follows. Section II introduces the main contributions regarding DIFT solutions. The threat model and assumptions are described in Section III. Our proposed approach, ARMHEx, is presented in Section IV. ARMHEx design and its implementation are discussed in Section V. Then, implementation results are given in Section VI. Finally, Section VII gives some conclusions and future perspectives for ARMHEx.

II. Related work

In order to overcome high time overheads of software solutions for DIFT (at least 300% [8]–[10]), hardware mechanisms were implemented. We can distinguish four main approaches:

1) **Filtering hardware accelerator** ([11], [12]). Instead of computing tags for each CPU instruction (as done in other approaches), this approach proposes to filter monitored events (e.g. system calls) before computing tags to lower DIFT time overhead.
2) **In-core** ([13], [14]). This approach relies on a deeply revised processor pipeline. Each stage of the pipeline is

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

Related work	Experimental Target	Hardcore portability	Communication interface	Interface simulated	Coprocessor isolated
Kannan et al. [2] Deng et al. [3],[4]	Softcore	No	Signals	No	No
Heo et al. [5] Davi et al. [6]	Softcore	Yes	System bus	No	No
Lee et al. [7]	Softcore	Yes	CDI(*Core Debug Interface*)	Yes	No
ARMHEx	Zynq SoC (ARM + FPGA)	Yes	EMIO (*Extended Multiplexed I/Os*) and System bus	No	Yes

duplicated with a hardware module in order to propagate tags all along the program execution. In [15], PUMP architecture modifies the CPU architecture to make DIFT computations in the processor pipeline. This in-core approach is not feasible with hardcore CPU such as ARM Cortex-A9 considered in this work.

3) **Offloading** ([16], [17]). In this case, DIFT operations are computed by a second general purpose processor. The required information for DIFT (i.e. PC register value, instruction encoding and `load/store` memory addresses) is sent by the processor running the application.
4) **Off-core** ([2]–[7]). 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 a softcore as in previous works) and the information required for DIFT is recovered through debug components, static analysis and instrumentation.

Table I compares features in existing works that are based on the same off-core approach as our proposed architecture ARMHEx. [2]–[4] implemented DIFT using a softcore processor. In these works, the CPU pipeline is modified 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 (such as ARM Cortex-A9 as considered in our work). The other related works mentioned in Table I are hardcore portable but present high time overheads due to the communication interface used between the CPU and the DIFT coprocessor.

In [5], [6], binary is instrumented to recover required information for DIFT. Instrumentation has the advantage of being flexible and portable on hardcore CPUs. However the overhead introduced by such a technique is too high: it may account for up to 86% of total DIFT performance overhead (as reported in [5]). To lower the overall DIFT performance penalty, the instrumentation overhead must be lowered.

Lee et al. [7] use CDI (*Core Debug Interface*) to extract information required for DIFT. Their considered debug component is ARM CoreSight ETM (*Event Trace Macrocell*) trace component which can provide information for each CPU instruction. For each instruction executed by the CPU, ETM generates an execution trace. This trace component has been replaced for performance purposes by CoreSight PTM (*Program Trace Macrocell*) which can provide information only on some instructions that modify the PC (*Program Counter*) register value. In this work, the main goal is to implement DIFT on a Xilinx Zynq SoC containing a PTM. The approach proposed in [7] cannot be used with PTM as all the information required for DIFT cannot be recovered. In addition to PTM, this work proposes to use static analysis and instrumentation to retrieve missing information.

No related work takes care of the DIFT coprocessor security. In existing works, components used by the DIFT coprocessor can be accessed from the CPU through unauthorized channels. We propose to secure ARMHEx by isolating it from the CPU and its memory. Furthermore, all implementations done in related works target a softcore CPU which explains their lack of deployment in industry. This work extends ideas presented in [18] and proposes the following contributions:

- It proposes to use CoreSight PTM, static analysis and binary instrumentation to recover information required for DIFT. It is shown that the proposed approach has a very low overhead mainly because PTM is a non-invasive component. Depending on the strategy, the instrumentation time overhead can be as low as 5.4% in average instead of more than 60% in related work.
- No existing work takes care of the DIFT coprocessor security. We propose to use ARM Trustzone to isolate ARMHEx in order to protect against unauthorized accesses.
- A proof-of-concept prototype and its implementation on Zynq SoC (Zedboard) is detailed. It is shown that the area and power overhead of proposed implementation is better than existing approaches.

III. Threat Model and Assumptions

This section describes the threat model and assumptions made for ARMHEx. Then, we explain how ARMHEx protects from data leakage.

A. Threat model

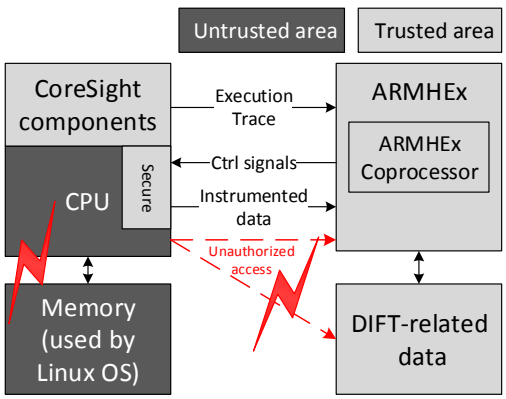

Fig. 1: Threat model

Figure 1 presents an overview schematic of an embedded system where ARMHEx is implemented. A default ARMHEx implementation is composed of a CPU (possibly multicore) and the extension ARMHEx which is implemented in the reconfigurable logic (both with a dedicated memory). We assume that our adversaries can launch a malicious application that contains software vulnerabilities. For instance, they can try to take advantage of buffer overflows, SQL injection, cross-site scripting or data leakage. Furthermore, adversaries may try to access the content of ARMHEx components or DIFT memory. For instance, an attacker can try mapping and writing to DIFT memory section used by ARMHEx. In this work, we consider that only following communication channels between CPU and ARMHEx are secure (*Execution trace*, *Control signals* and *instrumented data* in Figure 1). Furthermore, physical attacks are not considered (side-channel attacks, probing, JTAG attacks, memory attacks such as cold boot, EM injections...) in this work.

Our first goal is to prevent from user-space software attacks. ARMHEx uses DIFT to monitor information flows at execution time to prevent from these software attacks. Our second goal is to make sure that the specified trusted area is inaccessible from untrusted area through non-secure channels (colored in red in Figure 1). As an attacker may read or write to ARMHEx or its DIFT memory, reads/writes must not be allowed. ARMHEx uses ARM Trustzone to prevent from unauthorized access (writes or reads) to trusted area. We do not take into account any attacks originating from the secure world.

B. Example : DLP *(Data Leakage Prevention)*

As an introduction about DIFT implementation within the ARMHEx framework, an example of DLP is shown in Listing 1. It opens a file depending on user type (root or normal user), copies the file contents in a buffer and prints it. In order to avoid data leakage, the buffer should not be printed if it contains secret data as it may be the case if the user is root.

```
1  char buffer[20]; FILE *fs;
2  if(geteuid() != 0){ // user
3    fs = fopen("welcome", "r"); //public
4    if(!fs) exit (1);}
5  else{ // root
6    fs = fopen("passwd", "r"); //secret
7    if(!fs) exit(1);}
8  fread(buffer, 1, sizeof(buffer), fs);
9  fclose(fs);
10 printf("Buffer Value: %s \n", buffer);
```

Listing 1: Example code for DLP

ARMHEx operates on assembly instructions which facilitates DIFT implementation for all programming languages. The example code shown in Listing 1 is compiled for Zynq SoC to obtain the assembly code which is stored in memory used by Linux OS. System calls (e.g. `write` system call that is called for `printf` function call) need to be modified to send tag related information to ARMHEx coprocessor. For this purpose, an OS (*Operating System*) integrating support for DIFT such as Blare [19] will be considered in future works.

The first operation in order to implement DIFT is tag initialization. On line 1 of Listing 1, the tags of `buffer` and `fs` need to be initialized. Two levels of security (private and public) are considered in this work. The OS sends the tag value, address and size of `buffer` to the ARMHEx coprocessor. The allocated space for `buffer` is marked with a tag that can be of different size. The `fs` FILE pointer will be initialized with a tag according to the executed branch (if or else). The information about which branch is executed is obtained using PTM. If the user is not root, lines 3 and 4 are executed. In this case, the tag of `fs` is set to public as `welcome` file is public. Otherwise, lines 6 and 7 are executed. As `passwd` file is considered as secret, the tag of `fs` is set to secret.

On line 8, the tag of `buffer` is set to the tag of `fs` which can either be public or secret depending on the executed branch (if or else). This operation is performed by ARMHEx coprocessor. The tag check operation happens on line 10 when `buffer` is sent outside to the standard output. ARMHEx coprocessor sends tag value of `buffer` to the CPU: if it is secret, then a violation has occurred (a secret information is being sent outside the system) and an exception is raised. This example shows how ARMHEx uses the three operations required to implement DIFT described in Section I. Tag initialization and tag check operations need OS support while ARMHEx coprocessor alone is responsible for tag propagation.

IV. Proposed Approach

A DIFT implementation is efficient when required information is obtained in the shortest possible time. ARMHEx coprocessor requires at least three pieces of information to compute tags propagation:

1) PC register value.
2) Instruction encoding.
3) `load`/`store` memory addresses.

PC register value and some memory addresses are partially recovered using CoreSight components. Missing information about memory addresses and instruction encoding is obtained through static analysis and instrumentation.

A. Recovering information required for DIFT

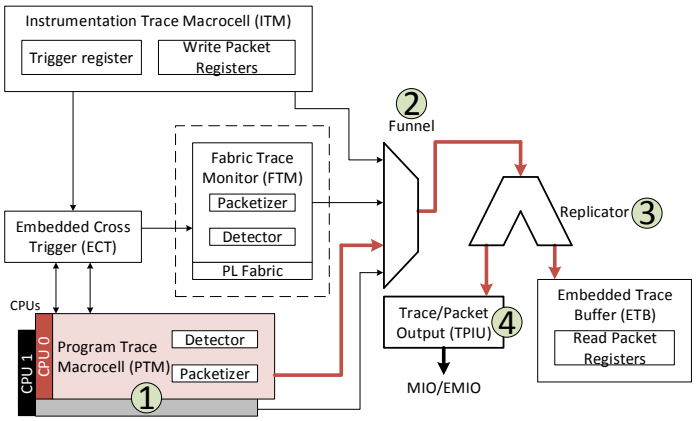

Fig. 2: CoreSight components in Xilinx Zynq

*1) **CoreSight components**:* CoreSight components (Figure 2) 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 and so on). ARMHEx uses these components to retrieve information on some instructions committed by the CPU at runtime. The most important classes of CoreSight components are detailed in following paragraphs.

a) Source: The PTM (1) generate execution traces for each waypoint (instructions that modify the program flow). A waypoint can be any of the following instructions: any indirect branches, conditional and unconditional direct branches, all exceptions, any instruction that changes the instruction set state or security state of the processor. Each time a waypoint occurs, the PTM generates a trace describing this waypoint. Table II shows an example of a trace output. The trace always starts with synchronization packets `A-Sync` and `I-Sync`. Then, each time a waypoint instruction is executed, a trace is generated. For instance, `bl` and `beq` instructions generate trace packets. If a `BAP` packet is generated, it means that the branch was taken. Otherwise, an `Atom` packet is generated. The PTM generates 11 different types of trace packets which are described in PFT (*Program Flow Trace*) specifications [20]. Packets relevant for ARMHEx are summed up in Table III.

TABLE II: Example code and corresponding trace

Address	Assembly code	Trace packets type
		`A-Sync`, `I-Sync`
860c	`sub r0, r1, r2`	-
8610	`bl 8480`	`BAP`
8614	`mov r3, r0`	-
8618	`ldr r3, [sp, #-8]`	-
861c	`cmp r3, #0`	-
8620	`beq 864c`	`BAP` or `Atom`
8624	`str r1, [r3, r2]`	-

The most important PTM feature used in ARMHEx is branch broadcasting which allows to obtain all branch addresses in the trace. The corresponding Linux driver did not implement this feature. We developed a patch that enables the use of branch broadcasting in Linux driver. The patch is currently under integration in the next Linux kernel release. Furthermore, PTM can be configured in different modes: it can trace all code or regions specified by start and end addresses. This feature is essential in ARMHEx as it needs traces of specific region of an application. For instance, we trace only `.text` section of a binary and its loaded libraries.

TABLE III: Relevant PFT packets in ARMHEx

Packet name	Packet Description
`A-Sync`	Alignment synchronization
`I-Sync`	Instruction synchronization (contains PC value)
`BAP`	Branch address packet (contains branch address)
`Atom`	Precise whether a branch was taken or not taken

b) Link: Funnel (2) and Replicator (3) are two components included in Xilinx Zynq SoCs: they transport the trace between source and sink components. Funnel takes care of merging traces from multiple sources into a single one: if both Cortex-A9 cores need to be traced, both PTMs must be enabled; furthermore, a source ID is assigned in order to identify each core from the Funnel point of view: it allows multicore debug tracing [21]. The Replicator duplicates the trace sent by Funnel and forwards it to sink components.

c) Sink: Traces can be transmitted to two components: ETB (*Embedded Trace Buffer*) is a small 4KB memory where traces can be stored (not used in this work); otherwise, TPIU (*Trace Port Interface Unit* (4)) is an entity able to export traces towards the reconfigurable logic.

In order to make sure that each change in the program flow is contained in trace sent by PTM, conditional execution must be disabled for instructions other than branches. Almost all instructions in ARM instruction set can be executed conditionally and can alter the program flow. However, these instructions are not traced by PTM. As a result, a LLVM pass has been created to allow conditional execution only for branch instructions. All other conditional instructions (e.g. `addeq`) are converted to a conditional branch (`bne`) followed by a normal instruction (`add`). This way, each change in program flow is contained in the execution trace generated by PTM. In other words, ARMHEx can determine which basic block is currently being executed. However, it does not know which tags need to be propagated inside each basic block. Static analysis allows to recover such information by generating tag dependencies instructions.

*2) **Static analysis**:* Table II shows a sample code used to illustrate how static analysis is used in ARMHEx. Instructions at address `860c`, `8614`, `8618`, `861c` and `8624` do not produce any trace. In order to recover information about all other instructions not contained in trace, the source code is statically analyzed before program execution. The static analysis will generate a tag dependencies instruction that must be executed by ARMHEx coprocessor.

TABLE IV: Example tag dependencies instructions

Example Instructions	Tag dependencies instructions
`sub r0, r1, r2`	$\underline{r0} = \underline{r1} + \underline{r2}$
`mov r3, r0`	$\underline{r3} = \underline{r0}$
`str r1, [PC, #4]`	$\underline{@Mem(PC+4)} = \underline{r1}$
`ldr r3, [SP, #-8]`	$\underline{r3} = \underline{@Mem(SP-8)}$
`str r1, [r3, r2]`	$\underline{@Mem(r3+r2)} = \underline{r1}$
(a)	**(b)**

Table IV shows the code which do not produce any trace (a) and corresponding tag dependencies (b). $\underline{r}$ is used to denote the tag of register `r`. For instance, for the first instruction in Table IV(a), the corresponding tag dependencies instruction is to associate tags of operands `r1` and `r2` towards the tag of destination register `r0`. Tag dependencies instructions are generated by analyzing the assembly representation. For each instruction, we determine operands type (register, memory or immediate) and their value. By using the types of operands, we can determine the information flows that takes place between the operands. For instance, if the first instruction of Table IV (a) is considered, we can determine that the three operands are registers. By determining registers number, we can determine the information flow. For all instructions that do not produce trace, we analyze instruction encoding and semantics to generate tag dependencies instructions. Furthermore, an instruction is added to keep track of SP value in ARMHEx. It is done for each CPU instruction that changes SP value directly (e.g. `sub SP, SP, #4`) or indirectly (e.g. `push`).

*3) **Instrumentation**:* In Table IV, `ldr` and `str` instructions contain memory addresses. These addresses need to be known to propagate their associated tags. There are three types of memory instructions :

(i) PC-relative (e.g. 3rd instruction of Table IV)
(ii) SP-relative (e.g. 4th instruction of Table IV)
(iii) Register-relative (e.g. 5th instruction of Table IV)

We can define two possible strategies to recover addresses contained in memory instructions. Both these strategies provide the same coverage as the related work instrumentation.

- **Strategy 1**. Each memory instruction is instrumented in order to send memory address(es) to ARMHEx coprocessor.
- **Strategy 2**. From all memory instructions, only register-relative instructions (iii) are instrumented. ARMHEx coprocessor knows the value of PC register thanks to the trace. Therefore, we do not need to instrument PC-relative memory instructions. Furthermore, the initial value of SP is sent to ARMHEx coprocessor when the program is launched which can keep track of SP value changes thanks to instructions inserted during static analysis. The SP-relative memory instructions do not need to be instrumented.

B. Security of the hardware extension

The overall architecture is shown in Figure 3. Existing works do not take into account the security of hardware DIFT extensions. It is important to protect hardware modules and memory sections from being modified through unauthorized channels. For instance, if a memory section used by ARMHEx coprocessor is modified by the software running on the Cortex-A9 core, ARMHEx coprocessor may produce false negatives or false positives. To avoid unauthorized access to ARMHEx, we propose to use ARM Trustzone to ensure that the CPU (untrusted part) cannot access components used by ARMHEx coprocessor. The Linux OS runs in non-secure world. In addition, the devices and memory used by Linux OS are declared as non-secure while ARMHEx and memory used by ARMHEx are declared as secure. When a non-secure element writes to a secure element, the operation will fail on check of NS (*Non-Secure*) bit. Similarly, if a non-secure element tries to read a secure element, the read will fail. ARM Trustzone is used to isolate ARMHEx with ARM CPU core.

V. ARMHEx DESIGN

ARMHEx is quite different from existing solutions, especially regarding how information needed for DIFT are recovered. This section explains the ARMHEx components used in this work as well as the operations done by the ARMHEx coprocessor.

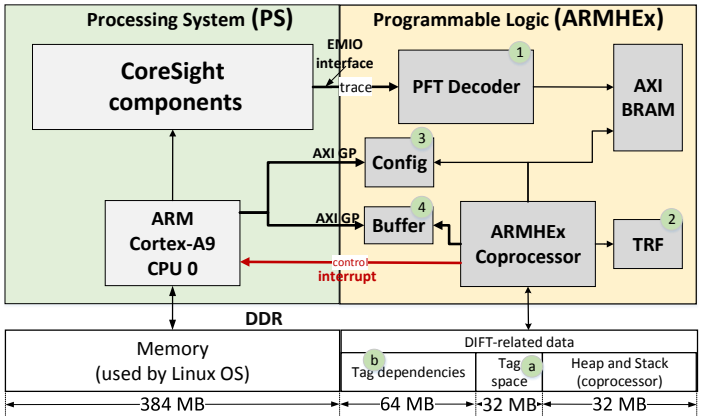

Fig. 3: Internal architecture of an ARMHEx-based system

A. ARMHEx components

The PFT decoder (1) is a state machine that decodes trace packets received from CoreSight components. As the trace is sent at 250 MHz by the TPIU, it needs to be decoded at the same frequency to avoid unnecessary storage overhead. The PTM sends different types of packets to analyze the code being executed on the CPU. Each type of packet has its own packet FSM (*Finite State Machine*) and a global state machine controls packet FSMs. Finally, decoded traces are stored in AXI Block RAM.

The TRF (*Tag Register File* (2)) is a register file that stores tags for each of 16 ARM CPU registers and 32 NEON registers. The Config IP (3) is an AXI slave IP containing a set of registers that provides a communication channel between the CPU and the ARMHEx coprocessor: it is used to configure tag propagation rules, send the initial value of SP and for debug purposes. Buffer (4) is a FIFO (AXI Slave write-only interface and a custom interface for read channel) that contains instrumented memory addresses.

Tags for memory addresses are stored in the tag space memory section (a). Tag dependencies ((b) in Figure 3) is also a memory section containing tag dependencies instructions obtained through static analysis (currently implemented with the Capstone disassembler API [22]). The structure of this memory section is shown in Figure 4 (inspired from a memory section layout in [5]). The upper part contains offset/jump addresses used to locate tag dependencies instructions related to basic blocks. At each basic block jump address, the header contains information such as the basic block size, information on how to decode the instructions. Then, there are instructions to be executed by the ARMHEx coprocessor.

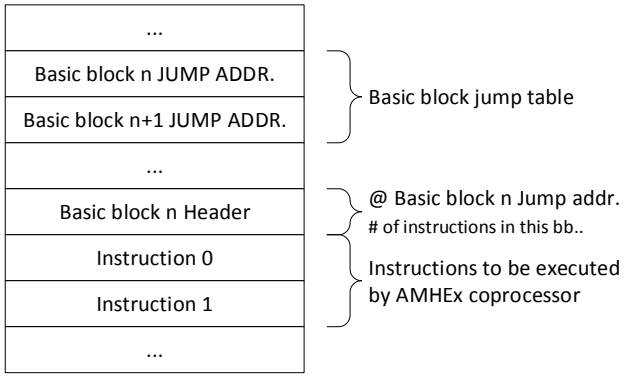

Fig. 4: Tag dependencies structure layout

B. ARMHEx operations

ARMHEx coprocessor computes tags for each tag dependencies instruction and checks the tags to detect a possible attack. Following steps are done to perform DIFT operations:

1) Reading decoded trace stored in AXI BRAM block.
2) Looking for corresponding basic block in tag dependencies section by reading the basic block jump table.
3) Reading basic block header, reading the tag dependencies instruction and decoding it.
4) For each tag dependencies instruction, looking for tags of source operands either in memory or TRF.
5) Computing tag of destination operand depending on current propagation rules stored in Config IP.
6) Updating corresponding tag in memory or TRF.
7) Checking for security policy violation and if a violation occurs raise an interruption.

The ARMHEx coprocessor, running at 100 MHz, need to be synchronized with the ARM CPU, running at 667MHz, in order to catch up. The synchronization is done on system calls as in previous works.

VI. Implementation results

Implementations were done with Vivado 2016.4 tools on a Xilinx Zedboard including a Z-7020 SoC (dual-core Cortex-A9 running at 667MHz and an Artix-7 FPGA). The FPGA logic has around 85K logic cells and 560 KB of Block RAMs. The ARMHEx coprocessor is implemented in a Microblaze softcore for this proof-of-concept. The goal of our evaluation is to prove the following points:

- Proposed strategies allow to recover required information for DIFT and reduce instrumentation time overhead (as this is the major factor of slowdown in existing works DIFT implementation).
- Security level provided by ARMHEx.
- Efficiency of ARMHEx compared to related works.

A. Time overhead analysis

TABLE V: Average execution time for custom benchmark

MiBench applications	CoreSight components disabled (in ms)	CoreSight components enabled (in ms)
bitcount	2.11	2.12
susan	29.29	29.35
jpeg	12.74	12.60
dijkstra	355.51	355.18
patricia	177.92	176.94
blowfish	13.94	13.87
rijndael	37.18	37.12
sha	117.48	117.88
CRC32	4132.31	4131.62
FFT	5.75	5.82
stringsearch	8.58	8.45

1) CoreSight components overhead: MiBench applications were tested with and without enabling CoreSight components. The negligible time difference observed in Table V is introduced by non-deterministic events (such as context switches). Verge et al. [23] showed that an execution time overhead could occur if traces are stored in the ETB. In this work, as shown in Table V, the time overhead of CoreSight components is negligible for two main reasons. First of all, the CoreSight PTM trace component is non-intrusive as it operates on a list of committed CPU instructions in parallel. The second reason is due to the configuration of CoreSight components: TPIU is used as a trace sink.

2) Static analysis: The static analysis does not add execution time overhead as it is performed before executing the application. However, it adds a static binary size overhead that corresponds to the storage of dependencies (i.e. output of static analysis). The communication overhead of ARMHEx is only due to code instrumentation.

3) Instrumentation overhead: The instrumentation time overhead is proportional to the number of instrumented instructions. Figure 5 shows the percentage of instrumented instructions over different strategies. In [5], in order to recover memory addresses, each branch instruction (`b/beq/bne/...`, `bl/blx`) as well as memory instructions are instrumented. Furthermore, another instruction is added for each direct branch in order to detect changes in the program flow. This strategy is referred as *Related work instrumentation* in Figure 5.

In this work, there is no need to instrument branch instructions thanks to CoreSight components. Then, we have two strategies.

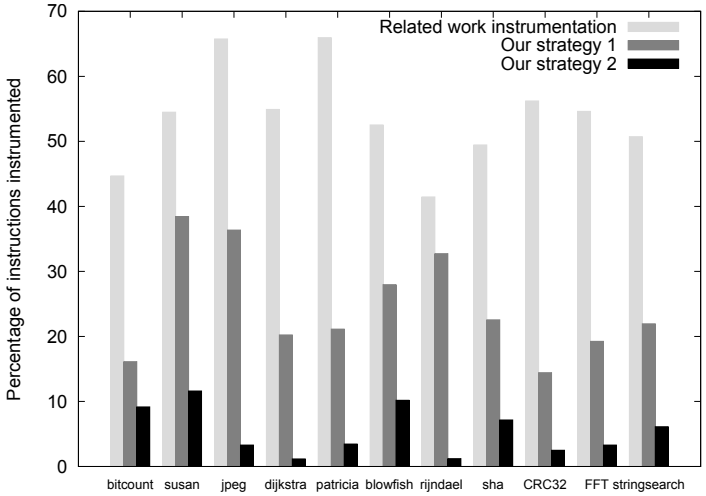

Fig. 5: Instrumentation overhead

Strategy #1 consists in instrumenting all memory instructions while strategy #2 only takes care of memory instructions relative to registers. For both strategies, instructions are added to send needed information to the buffer implemented in the reconfigurable logic.

The average time overhead for strategy #1 is 24.6% while it reaches 53.7% for related work instrumentation strategy. The average time overhead for strategy #2 is 5.37% which is better than the overhead of 60% reported by Heo et al. [5]. For some applications (such as `dijkstra`, `rijndael` or `CRC32`), the instrumentation overhead (for strategy #2) is less than 2%: it is due to the fact that there are few register-relative memory instructions in these applications.

B. Area

Area results are shown in Table VI. Most of the FPGA area is filled by the AXI interconnect (5.87%), Config IP (5.20%) and the Microblaze softcore (4.62%). Other IPs occupy less than 1% of the FPGA area in terms of slices.

TABLE VI: Area results of ARMHEx on Xilinx Zynq Z-7020

IP Name	Slice LUTs	Slice Registers	Slice (in %)	BRAM Tile
Microblaze	1578	1407	614 (4.62)	6
MDM	102	110	40 (0.30)	0
Local memory	14	4	11 (0.08)	32
PFT Decoder	105	211	60 (0.45)	0
AXI TRF	53	105	24 (0.18)	1
Config	914	2141	692 (5.20)	0
AXI Interconnect	1788	2436	781 (5.87)	0
BRAM	2	0	1 (0.01)	2
BRAM Controller	157	168	59 (0.44)	0
Miscellaneous	641	586	171 (1.29)	0
Total Design	5354 (10.06%)	7168 (6.74%)	2453 (18.44%)	41 (29.29%)
Total Available	**53200**	**106400**	**13300**	**140**

In this work, ARMHEx targets a single Cortex-A9 core. Implementation results show that a Cortex-A9 dual-core, such as the one included in the Zynq Z-7020, could be easily protected. In the current configuration, the ARMHEx infrastructure could cover up to 5 Cortex-A9 cores simultaneously.

C. Security evaluation

To evaluate ARMHEx, we tested the example presented in Section III-B. When the application is run as normal user, the program terminates correctly. However, when it is run as root user, ARMHEx coprocessor interrupts CPU to stop the application. It shows that ARMHEx can detect security policy violations.

SafeG [24] dual-OS monitor has been used to take profit of ARM Trustzone. The Linux runs in normal world and the TOPPERS/FMP kernel ([25]) runs in secure world. In order to evaluate ARMHEx isolation, we tried to access secure memory region used by ARMHEx coprocessor from Linux (user space), the access was refused due to a bus error. It shows that unlike existing approaches, ARMHEx do not allow unauthorized accesses.

D. Comparison with previous works

TABLE VII: Performance comparison with related work

Approaches	Kannan [2]	Deng [3]	Heo [5]	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

Table VII shows a performance comparison of ARMHEx with previous off-core approaches. Unlike previous works, ARMHEx is 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 [7] are not portable on Zynq SoC due to CoreSight PTM component. Furthermore, the time cost for communication between a CPU and the coprocessor is 5.4% in this work compared to 60% in [5]. In terms of area, ARMHEx has the best coprocessor/processor ratio: the reason is that the CPU used in our work is a Cortex-A9 which has around 26 million gates [26]. Other works use softcores as their main CPU which have a lower number of gates. If we compare area results (in terms of gate counts), our approach still performs better. Moreover, regarding the power ratio of a DIFT-enhanced architecture, ARMHEx is better than [5] (16 % instead of 24%). However, Deng et al. [3] have a better power overhead than ARMHEx because they implemented a dedicated hardware module for DIFT instead of a coprocessor-based approach as in ARMHEx. ARMHEx is able to operate at a maximum frequency of 250 MHz (bridled at 100 MHz for the first implementation because of a Microblaze used for DIFT computations).

VII. 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 along with static analysis and instrumentation to recover required information for DIFT. It is shown that by using our approach, only 6% of instructions need to be instrumented in an application compared to 60% instrumented instructions in related works. Furthermore, we propose to protect ARMHEx and its components by isolating them using ARM Trustzone. Our proposed approach ARMHEx has been implemented on a Zynq SoC. Implementation results show interesting perspectives for ARMHEx in terms of multicore runtime security. ARMHEx can be implemented in parallel as it has a moderate impact in terms of area (less than 20% of FPGA area is currently used).

Acknowledgments

This work is realized in the context of the HardBlare project funded by the French Labex CominLabs and Brittany region.

References

[1] S. Moore and S. Chong, "Static analysis for efficient hybrid information-flow control," in *2011 IEEE 24th Computer Security Foundations Symposium*, June 2011, pp. 146–160.

[2] H. Kannan, M. Dalton, and C. Kozyrakis, "Decoupling dynamic information flow tracking with a dedicated coprocessor," in *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, June 2009.

[3] 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 *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43, 2010.

[4] D. Y. Deng and G. E. Suh, "High-performance parallel accelerator for flexible and efficient run-time monitoring," in *Proceedings of the 2012 42Nd Annual IEEE/IFIP DSN*, ser. DSN '12, 2012, pp. 1–12.

[5] 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 Trans. Des. Autom. Electron. Syst.*, vol. 20, no. 4, pp. 53:1–53:32, Sep. 2015.

[6] L. Davi, M. Hanreich, D. Paul, A. R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin, "Hafix: Hardware-assisted flow integrity extension," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*.

[7] J. Lee, I. Heo, Y. Lee, and Y. Paek, "Efficient security monitoring with the core debug interface in an embedded processor," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 22, no. 1, pp. 8:1–8:29, May 2016.

[8] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," 2005.

[9] L. C. Lam and T. c. Chiueh, "A general dynamic information flow tracking framework for security applications," in *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, Dec 2006.

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

[11] S. Fytraki, E. Vlachos, Y. O. Koçberber, B. Falsafi, and B. Grot, "FADE: A programmable filtering accelerator for instruction-grain monitoring," in *20th IEEE HPCA, HPCA 2014, Orlando, FL, USA*.

[12] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos, "Flexible hardware acceleration for instruction-grain program monitoring," *SIGARCH Comput. Archit. News*, Jun. 2008.

[13] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: A flexible information flow architecture for software security," *SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 482–493, Jun. 2007.

[14] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "Flexitaint: A programmable accelerator for dynamic taint propagation," in *2008 IEEE 14th HPCA*, Feb 2008, pp. 173–184.

[15] U. Dhawan, C. Hritcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight, Jr., B. C. Pierce, and A. DeHon, "Architectural support for software-defined metadata processing," *SIGARCH Comput. Archit. News*, vol. 43, no. 1, pp. 487–502, Mar. 2015. [Online]. Available: http://doi.acm.org/10.1145/2786763.2694383

[16] V. Nagarajan, H.-S. Kim, Y. Wu, and R. Gupta, "Dynamic information flow tracking on multicores," *Workshop on Interaction between Compilers and Computer Architectures, (colocated with HPCA)*, feb 2008.

[17] K. Jee, V. P. Kemerlis, A. D. Keromytis, and G. Portokalidis, "Shadowreplica: Efficient parallelization of dynamic data flow tracking," in *Proceedings of CCS*. New York, NY, USA: ACM, 2013, pp. 235–246.

[18] M. Abdul Wahab, P. Cotret, M. Nasr Allah, G. Hiet, V. Lapotre, and G. Gogniat, "Towards a hardware-assisted information flow tracking ecosystem for ARM processors," in *FPL 2016*, Lausanne, Switzerland, Aug. 2016. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01337579

[19] L. George *et al.*, "Blare tools: A policy-based intrusion detection system automatically set by the security policy," in *RAID 2009*, 2009.

[20] ARM, *CoreSight Program Flow Trace - Architecture specifications*, 2011, http://goo.gl/S0Zt5V.

[21] ARM, *CoreSight Components - Technical Reference manual*, 2009, http://goo.gl/hm9X6Q.

[22] Capstone disassembly library. [Online]. Available: http://www.capstone-engine.org/

[23] A. Vergé, "Hardware-assisted software tracing," in *Embedded Linux Conference (ELC)*, April 2014.

[24] SafeG dual-OS monitor. [Online]. Available: http://www.toppers.jp/en/safeg.html

[25] TOPPERS/FMP Kernel. [Online]. Available: http://www.toppers.jp/en/fmp-kernel.html

[26] Olle Svanfeldt-Winter, Lafond Sébastien, and Lilius Johan, "Evaluation of the energy efficiency of arm based processors for cloud infrastructure," Turku Centre for Computer Science, Tech. Rep.