A hardware/software co-design approach for security analysis of application behavior
Vianney Lapotre

To cite this version:

HAL Id: hal-02013870
https://hal.archives-ouvertes.fr/hal-02013870
Submitted on 11 Feb 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
A hardware/software co-design approach for security analysis of application behavior
Applications on Dynamic Information Flow Tracking

Vianney Lapôtre

Université Bretagne Sud - Lab-STICC, vianney.lapotre@univ-ubs.fr

January 23, 2019
HardBlare project
Started in October 2015.

Partners (all from Brittany !)

- IETR/CentraleSupélec (SCEE) @ Rennes
  - Pascal Cotret (Ass. Prof.) now engineer at Thales
  - Muhammad Abdul Wahab (PhD student)
- IRISA/CentraleSupélec/Inria (CIDRE) @ Rennes
  - Guillaume Hiet (Ass. Prof.)
  - Mounir Nasr Allah (PhD student)
- Lab-STIC/CUBS @ Lorient
  - Guy Gogniat (Full Prof.), Vianney Laptre (Ass. Prof.)
  - Arnab Kumar Biswas (Postdoc)

What do we do in this project ?

Hardware extensions for DIFT/DIFC (Dynamic Information Flow Tracking / Dynamic Information Flow Control) on embedded processors
Data security: principles

**Principles**
- Confidentiality
- Integrity
- Availability

**Security Policy**
- Which security property is expected on each information container (file, variable, register, etc.)?
- What operations are allowed on each container?
## Threat model

- Side-channel attacks not taken into account
- Software attacks: buffer overflow, ROP...
Software security: Existing solutions

Security mechanisms
Detect, prevent or recover from a security attack

Preventive mechanisms
Enforce the security policy:
- Cryptographic mechanisms
- Isolation (e.g., Trustzone, SAM L11)
- Formal proof, etc.

Reactive mechanisms
Monitor the system and detect any security policy violation to recover
- Intrusion detection systems (e.g., Snort, OSSEC)
  - Dynamic Information flow tracking (DIFT)
Dynamic Information Flow Tracking (DIFT)

Motivation

**DIFT for security purposes**: Integrity and Confidentiality

**DIFT principle**

- We attach **labels** called tags to **containers** and specify an information flow **policy**, i.e. relations between tags.
- At runtime, we **propagate** tags to reflect information flows that occur and **detect** any **policy violation**.

![Diagram showing information flow between different files and channels with labels indicating security levels.](Image)
Dynamic Information Flow Tracking (DIFT)

Three steps

- Tag initialization
Dynamic Information Flow Tracking (DIFT)

Three steps

- Tag initialization
- Tag propagation
Dynamic Information Flow Tracking (DIFT)

**Three steps**

- Tag initialization
- Tag propagation
- Tag check
Dynamic Information Flow Tracking (DIFT)

Three steps
- Tag initialization
- Tag propagation
- Tag check

Levels of IFT
- Application level
Dynamic Information Flow Tracking (DIFT)

**Three steps**
- Tag initialization
- Tag propagation
- Tag check

**Levels of IFT**
- Application level
- OS level
Dynamic Information Flow Tracking (DIFT)

Three steps

- Tag initialization
- Tag propagation
- Tag check

Levels of IFT

- Application level
- OS level
- Low level
OS-level Software DIFT (coarse-grained)

Description

- Monitor is implemented within the OS kernel
- Information flows = system calls

Related Work

- Dedicated OS\textsuperscript{1} : Asbestos, HiStar, Flume
- Modification of existing OS : \textbf{Blare}\textsuperscript{2}

Pros & Cons

+ Small runtime overhead (< 10%)
+ Kernel space isolation (hardware support) helps protecting the monitor
  - Overapproximation issue

\textsuperscript{1}Eal.05; Zal.06a; Kal.07.
\textsuperscript{2}Gal.11; HF12.
Application-level Software DIFT (medium and fine-grained)

### Description
- Monitors are implemented within each application
- Information flows = affectations + conditional branching

### Related Work
- Machine code
- Specific language

### Pros & Cons
- Gain in precision (hybrid analysis, SME, faceted values)
- Huge overhead (x3 to x37)
- Few or no isolation: the monitor needs to protect itself

---

3 NS05; HJR10.
4 CF07; Nal.07.
Hardware-based DIFT (fine-grained)

Figure: In-core DIFT ⁵

Figure: Dedicated CPU for DIFT ⁶

⁵ raksha’07.
⁶ Vijay08.
Hardware-based DIFT (fine-grained)

Figure: Dedicated DIFT co-processor

\[7\]

\[7\text{raksha}^\circ \text{09}.\]
Fine-grained DIFT: comparison of the existing approaches

<table>
<thead>
<tr>
<th></th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software</td>
<td>Flexible security policies</td>
<td>Runtime overhead (from 300% to 3700%)</td>
</tr>
<tr>
<td>In-core DIFT</td>
<td>Low overhead (&lt;10%)</td>
<td>Invasive modifications</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Few security policies</td>
</tr>
<tr>
<td>Dedicated CPU for DIFT</td>
<td>Low overhead (&lt;10%)</td>
<td>Wasting resources</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Energy consumption (x 2)</td>
</tr>
<tr>
<td>Dedicated DIFT coprocessor</td>
<td>Flexible security policies</td>
<td>Communication between CPU and DIFT Coprocessor</td>
</tr>
<tr>
<td></td>
<td>Low runtime overhead (&lt;10%)</td>
<td>CPU not modified</td>
</tr>
</tbody>
</table>
DIFT Example: Memory corruption
Attacker overwrites return address and takes control

```c
int idx = tainted_input; //stdin (> BUFFER SIZE)
buffer[idx] = x; // buffer overflow
```

```
set r1 ← &tainted_input
load r2 ← M[r1]
add r4 ← r2 + r3
store M[r4] ← r5
```

pseudo-code

<table>
<thead>
<tr>
<th>T</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>r1</td>
</tr>
<tr>
<td></td>
<td>r2</td>
</tr>
<tr>
<td></td>
<td>r3: &amp;buffer</td>
</tr>
<tr>
<td></td>
<td>r4</td>
</tr>
<tr>
<td></td>
<td>r5: x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Return Address</td>
</tr>
<tr>
<td></td>
<td>int buffer[Size]</td>
</tr>
</tbody>
</table>
DIFT Example: Memory corruption
Attacker overwrites return address and takes control

```c
int idx = tainted_input; //stdin (> BUFFER SIZE)
buffer[idx] = x; // buffer overflow
```

pseudo-code

<table>
<thead>
<tr>
<th>T</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>r1:&amp;input</td>
</tr>
<tr>
<td></td>
<td>r2</td>
</tr>
<tr>
<td></td>
<td>r3:&amp;buffer</td>
</tr>
<tr>
<td></td>
<td>r4</td>
</tr>
<tr>
<td></td>
<td>r5:x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Return Address</td>
</tr>
<tr>
<td></td>
<td>int buffer[Size]</td>
</tr>
</tbody>
</table>
DIFT Example: Memory corruption
Attacker overwrites return address and takes control

```c
int idx = tainted_input; //stdin (> BUFFER SIZE)
buffer[idx] = x; // buffer overflow
```

```plaintext
<table>
<thead>
<tr>
<th>T</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>r1:&amp;input</td>
</tr>
<tr>
<td></td>
<td>r2:idx=input</td>
</tr>
<tr>
<td></td>
<td>r3:&amp;buffer</td>
</tr>
<tr>
<td></td>
<td>r4</td>
</tr>
<tr>
<td></td>
<td>r5:x</td>
</tr>
</tbody>
</table>
```

pseudo-code

```plaintext
set r1 ← &tainted_input
load r2 ← M[r1]
add r4 ← r2 + r3
store M[r4] ← r5
```

<table>
<thead>
<tr>
<th>T</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Return Address</td>
</tr>
<tr>
<td></td>
<td>int buffer[Size]</td>
</tr>
</tbody>
</table>
DIFT Example: Memory corruption
Attacker overwrites return address and takes control

```c
int idx = tainted_input; // stdin (> BUFFER SIZE)
buffer[idx] = x; // buffer overflow
```

```
set r1 ← &tainted_input
load r2 ← M[r1]
add r4 ← r2 + r3
store M[r4] ← r5
```

pseudo-code

<table>
<thead>
<tr>
<th>T</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>r1:&amp;input</td>
</tr>
<tr>
<td></td>
<td>r2:idx=input</td>
</tr>
<tr>
<td></td>
<td>r3:&amp;buffer</td>
</tr>
<tr>
<td></td>
<td>r4:&amp;buffer+idx</td>
</tr>
<tr>
<td></td>
<td>r5:x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Return Address</td>
</tr>
<tr>
<td></td>
<td>int buffer[Size]</td>
</tr>
</tbody>
</table>
DIFT Example: Memory corruption
Attacker overwrites return address and takes control

```c
int idx = tainted_input; // stdin (> BUFFER SIZE)
buffer[idx] = x; // buffer overflow
```

```plaintext
data T
r1: &tainted_input
r2: idx = input
r3: &buffer
r4: &buffer + idx
r5: x
```

```plaintext
set r1 ← &tainted_input
load r2 ← M[r1]
add r4 ← r2 + r3
store M[r4] ← r5
```

table T Data
Return Address
int buffer[Size]
HardBlare approach

Objectives

- Combine hardware level and OS level approaches
- Design and implement a realistic proof-of-concept
  - Unmodified (ASIC) main CPU (related work rely on softcores)
  - Dedicated DIFT coprocessor on FPGA
  - Rely on existing OS and applications (Linux system)

Technological choices

- Xilinx Zynq SoC (2 cores ARM Cortex A9 + FPGA)
- Dedicated Linux distribution using Yocto

Challenge

Semantic gap: limited visibility of CPU instructions on FPGA side
Information required for DIFT

**Hypothesis:** Application with source code

- Program Counter (PC)
- Instruction encoding
- Memory addresses
- Tags of files
What can I do with my processor?

- CoreSight: debug components
- Available in most of Cortex-A + Cortex-M3 (for ARM)
- Can export debug-related infos
CoreSight components
Coresight PTM

Features

- **Trace Filter** (all code or regions of code)

<table>
<thead>
<tr>
<th>ELF Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program header table</td>
</tr>
<tr>
<td>Section 1</td>
</tr>
<tr>
<td>Section 2</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Section n</td>
</tr>
<tr>
<td>Section header table</td>
</tr>
</tbody>
</table>
Coresight PTM

Features

- **Trace Filter** (all code or regions of code)
- **Branch Broadcast** \(^8\)

---

\(^{8}\)Linux driver for PTM patched to support Branch broadcast feature
Coresight PTM

Features

- **Trace Filter** (all code or regions of code)
- **Branch Broadcast**
- Context ID comparator

(i) MOV PC, LR
(ii) ADD R1, R2, R3
(iii) B 0x8084

---

8Linux driver for PTM patched to support Branch broadcast feature
Example Trace

Source code

```c
int i;
for (i = 0; i < 10; i++)
```

Example Trace

Source code

```
int i;
for (i = 0; i < 10; i++)
```

Assembly

```
8638 for_loop:
...
b 8654:
...
866c: bcc 8654
```
Example Trace

Source code

```c
int i;
for (i = 0; i < 10; i++)
```

Assembly

```assembly
8638 for_loop:
...
8654:
...
866c: bcc 8654
```

Trace

```
00 00 00 00 00 80 08 38 86 00 00
86 01 00 00 00 00 00 00 00 00
```
Example Trace

Source code

```c
int i;
for (i = 0; i < 10; i++)
```

Assembly

```assembly
8638 for_loop:
...
b 8654:
...
866c: bcc 8654
```

Trace

```
00 00 00 00 00 80 08 38 86 00 00
86 01 00 00 00 00 00 00 00 00 00
```

Decoded Trace

A-sync
Address 00008638, (I-sync Context 00000000, IB 21)
Address 00008654, Branch Address packet (x 10)
int main() {
    int file_public, file_secret, file_output;
    char public_buffer[1024];
    char secret_buffer[1024];
    char *temporary_buffer;
    file_public = open("files/public.txt",O_RDONLY);
    file_secret = open("files/secret.txt",O_RDONLY);
    file_output = open("files/output.txt",O_WRONLY);
    read(file_public, public_buffer, 1024);
    read(file_secret, secret_buffer, 1024);
    
    if( (rand() % 2) == 0 ){
        temporary_buffer = public_buffer;
    } else{
        temporary_buffer = secret_buffer;
    }
    
    write(file_output, temporary_buffer, 1024);
    return 0;
}
CoreSight components - Performance overhead

- Execution time measured with and without enabling CoreSight components
- No change in measured execution time
- Negligible runtime overhead
  1. PTM non-intrusive (dedicated HW module that works in parallel)
  2. Configuration of CoreSight components (TPIU sink used\(^9\) rather than ETB)

\(^9\)Linux driver for TPIU has been patched
Debug components on a hardcore CPU

Recovery of Program Counter

![Diagram showing debug components and their connections]

- **Debug components**
  - **Application**
  - **Hardcore CPU**
  - **Hard disk**
  - **Memory**

- **DIFT monitor**

- **Coprocessor**

**Trace**
Static Analysis

Problem

We need to know what’s happened between two jumps

Solution

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

Examples:

\[
\text{add } r0, r1, r2 \Rightarrow r0 \leftarrow r1 \cup r2
\]

and \(r3, r4, r5 \Rightarrow r3 \leftarrow r4 \cup r5\)
Static Analysis

Recovery of instruction encoding
Instrumentation

- Some addresses are resolved/calculated at run-time:
  - Solution: instrument the code
- The instrumentation is done during the last phase of the compilation process.
- The register **r9 is dedicated** for the instrumentation.
- The instrumentation FIFO address is retrieved via a **UIO Driver**.

**Examples:**

- \texttt{ldr r0, [r2]} \Rightarrow \texttt{str r2, [r9]}
- \texttt{str r3, [r4]} \Rightarrow \texttt{str r5, [r9]}
- \texttt{ldr r0, [r2]}
  \texttt{str r3, [r5]}
Instrumentation

Recover memory addresses

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Annotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldr r1, [r2, #4]</td>
<td>r1 ← mem (r2 + 4)</td>
</tr>
</tbody>
</table>

Two possible strategies

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

Recover all memory address through instrumentation

### Example Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Annotations</th>
<th>Memory address recovery</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub r0, r1, r2</td>
<td>( r0 = r1 + r2 )</td>
<td>Instrumented</td>
</tr>
<tr>
<td>mov r3, r0</td>
<td>( r3 = r0 )</td>
<td>Instrumented</td>
</tr>
<tr>
<td>str r1, [PC, #4]</td>
<td>( @\text{Mem}(\text{PC}+4) = r1 )</td>
<td>Instrumented</td>
</tr>
<tr>
<td>ldr r3, [SP, #-8]</td>
<td>( r3 = @\text{Mem}(\text{SP}-8) )</td>
<td>Instrumented</td>
</tr>
<tr>
<td>str r1, [r3, r2]</td>
<td>( @\text{Mem}(r3+r2) = r1 )</td>
<td>Instrumented</td>
</tr>
</tbody>
</table>
Recover only register-relative memory address through instrumentation

<table>
<thead>
<tr>
<th>Example Instructions</th>
<th>Annotations</th>
<th>Memory address recovery</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub r0, r1, r2</td>
<td>r0 = r1 + r2</td>
<td>CoreSight PTM</td>
</tr>
<tr>
<td>mov r3, r0</td>
<td>r3 = r0</td>
<td>Static analysis</td>
</tr>
<tr>
<td>str r1, [PC, #4]</td>
<td>@Mem(PC+4) = r1</td>
<td>Instrumented</td>
</tr>
<tr>
<td>ldr r3, [SP, #-8]</td>
<td>r3 = @Mem(SP-8)</td>
<td></td>
</tr>
<tr>
<td>str r1, [r3, r2]</td>
<td>@Mem(r3+r2) = r1</td>
<td></td>
</tr>
</tbody>
</table>
**Instrumentation time overhead**

- **Figure**: Normalized execution time of MiBench benchmark for different strategies.

- **Related work**
  - Strategy 1: 53.7% (Original Program) 24.6% (Host instrument)
  - Strategy 2: 5.37% (Original Program) -90% (Host instrument)

- **Strategy 1**
  - Normalized Execution Time: 1.5

- **Strategy 2**
  - Normalized Execution Time: 1.0
Instrumentation

Recovery of memory addresses

Diagram showing the flow of data from Debug components, Application, Hardcore CPU, Hard disk, Memory, Instrumentation, Trace Memory, and Coprocessor.
RfBlare: System calls

- **Problem:** We want to transmit tags from/to the operating system.
  *Solution:* Linux Security Modules Hooks

- **Problem:** We want to persistently store tags in the system.
  *Solution:* Extended file attributes

- When **reading** data from a file:
  *We are propagating the tag of the read file to the destination buffer.*

- When **writing** data to a file:
  *We are propagating the tag of the source buffer to the destination file.*
System calls: RFBlaire

Recovery of tags of files
Global architecture

**Modified Linux**

- **Application**
- **CS components**
- **ARM Cortex-A9**
- **Tag of files**

**PL: Programmable Logic**

- **Hard disk**
- **CPU memory (384 MB)**
- **Tag annotations (64 MB)**
- **Tag memory (64 MB)**

**Diagram:**
- AXI interconnect
- Instrumentation
- Kernel2 monitor
- Monitor2 kernel
- DIFT coprocessor
- PFT decoder
- Decoded trace memory
- DDR
- Interrupt

**Components:**
- AXI interconnect
- Instrumentation
- Kernel2 monitor
- Monitor2 kernel
- DIFT coprocessor
- PFT decoder
- Decoded trace memory
- DDR

**Tag annotations (64 MB)**

**Tag memory (64 MB)**

**Hard disk**

**CPU memory (384 MB)**
Dedicated DIFT coprocessor
DIFT coprocessor - Internal architecture

- Dispatcher fully pipelined (classical MIPS ISA)
- TMC (Tag Management Core) pipelined (custom ISA)
  - Compile-time security policy
  - Runtime security policy
Use cases: Multiple security policies

Annotations memory → TMC (security policy 2) → Tag memory

Decoded trace memory → Dispatcher

Annotations memory (security policy 1) → TMC → Tag memory

BRAM

Tag annotations

DDR

Tag RR \(T_1, T_2\)
Use cases: Multiple processes/threads

- Decoded trace memory
- Dispatcher
- Tag annotations
- Tag memory

- Annotations memory
- TMC (process 2)
- DDR

- Annotations memory
- TMC (process 1)
- DDR
Extension for 2 threads - Trace details

Decoded trace | Context ID | Stored address
---|---|---
00010574 | 0004d2 42 | 00010574
00010428 | 0004d2 42 | 00010428
00010584 | 0004d2 42 | 00010584
000103c8 | 0004d2 42 | 000103c8
00010598 | 0004d2 42 | 00010598
000103f8 | 0004d2 42 | 000103f8
00010574 | 0004d3 42 | 00010575
00010428 | 0004d3 42 | 00010429
00010584 | 0004d3 42 | 00010585

Trace

A-sync | l-sync | Branch address packet
---|---|---
00 00 00 00 00 80 08 74 05 01 00 21 42 d2 04 00
95 04 08 84 05 01 00 21 42 d2 04 00 e5 03 08 98
05 01 00 21 42 d2 04 00 fd 03 08 74 05 01 00 21
42 d3 04 00 95 04 08 84 05 01 00 21 42 d3 04 00
Related work strategy
Strategy proposed by Heo et al.\textsuperscript{10} adapted with OS support

<table>
<thead>
<tr>
<th>Example Instructions</th>
<th>Annotations</th>
<th>Strategy 1</th>
<th>Strategy 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub r0, r1, r2</td>
<td>( r0 = r1 + r2 )</td>
<td>Instrumented</td>
<td>CoreSight PTM</td>
</tr>
<tr>
<td>mov r3, r0</td>
<td>( r3 = r0 )</td>
<td>Instrumented</td>
<td>Static analysis</td>
</tr>
<tr>
<td>str r1, [PC, #4]</td>
<td>( \text{@Mem(PC+4)} = r1 )</td>
<td>Instrumented</td>
<td>Instrumented</td>
</tr>
<tr>
<td>ldr r3, [SP, #-8]</td>
<td>( r3 = \text{@Mem(SP-8)} )</td>
<td>Instrumented</td>
<td>Instrumented</td>
</tr>
<tr>
<td>str r1, [r3, r2]</td>
<td>( \text{@Mem(r3+r2)} = r1 )</td>
<td>Instrumented</td>
<td>Instrumented</td>
</tr>
</tbody>
</table>

\textsuperscript{10}Heo\textsuperscript{\textsuperscript{\textsuperscript{15}}}.
Instrumentation time overhead (with OS support)

<table>
<thead>
<tr>
<th></th>
<th>Related work</th>
<th>Strategy 1</th>
<th>Strategy 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Program</td>
<td>12.79</td>
<td>10.43</td>
<td>3.35</td>
</tr>
<tr>
<td>Host instrument</td>
<td>-18%</td>
<td>-74%</td>
<td>-74%</td>
</tr>
</tbody>
</table>

-18% decrease for Strategy 1
-74% decrease for Strategy 2
## Comparison with related works

<table>
<thead>
<tr>
<th>Approaches</th>
<th>Kannan</th>
<th>Deng</th>
<th>Heo</th>
<th>Hardblare</th>
<th>Heo adapted</th>
<th>Hardblare</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area overhead</td>
<td>6.4%</td>
<td>14.8%</td>
<td>14.47%</td>
<td>0.47%</td>
<td>N/A</td>
<td>0.95%</td>
</tr>
<tr>
<td>Power overhead</td>
<td>N/A</td>
<td>6.3%</td>
<td>24%</td>
<td>8.45%</td>
<td>N/A</td>
<td>16.2%</td>
</tr>
<tr>
<td>Max frequency</td>
<td>N/A</td>
<td>256 MHz</td>
<td>N/A</td>
<td>250 MHz</td>
<td>N/A</td>
<td>250 MHz</td>
</tr>
<tr>
<td>Communication time overhead</td>
<td>N/A</td>
<td>N/A</td>
<td>60%</td>
<td>5.4%</td>
<td>1280%</td>
<td>335%</td>
</tr>
<tr>
<td>Hardcore portability</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Main CPU</td>
<td>Softcore</td>
<td>Softcore</td>
<td>Softcore</td>
<td>Hardcore</td>
<td>Hardcore</td>
<td>Hardcore</td>
</tr>
<tr>
<td>Library instrumentation</td>
<td>N/A</td>
<td>N/A</td>
<td>partial</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>FP support</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Multi-threaded support</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Conclusion and Perspectives

Take away:

- CoreSight PTM allows to obtain runtime information (Program Flow)
- Non-intrusive tracing ⇒ Negligible performance overhead
- Isolation of hardware IPs with ARM Trustzone
- Integration of OS support in the hardware-assisted DIFT
- Implementation of the proposed approach on the Zynq SoC
- Scalable solution for multiple security policies and multicore/multiprocessor systems

Perspectives:

- Full PoC later this year (SoC files + Yocto)
- Intel / ST? (study)
- Multicore multi-thread IFT
A hardware/software co-design approach for security analysis of application behavior
Applications on Dynamic Information Flow Tracking

Vianney Lapôtre
Université Bretagne Sud - Lab-STICC, vianney.lapotre@univ-ubs.fr

January 23, 2019

Many thanks to Muhammad, Mounir, Arnab, Pascal, Guillaume and Guy :)

https://hardblare.cominlabs.u-bretagneloire.fr
Bibliography I


**Vijay08** Dynamic Information Tracking on Multicores.

**raksha’09** Decoupling dynamic information flow tracking with a dedicated coprocessor.

**harmoni’12** High-performance parallel accelerator for flexible and efficient run-time monitoring.

**pau’15** Implementing an Application-Specific Instruction-Set Processor for System-Level Dynamic Program Analysis Engines.

**fpl’17** ARMHEx: A hardware extension for DIFT on ARM-based SoCs.
Bibliography II

**Eal.05** Labels and event processes in the asbestos operating system.

**Zal.06a** Making information flow explicit in HiStar.

**Kal.07** Information Flow Control for Standard OS Abstractions.

**Gal.11** Information Flow Control for Intrusion Detection derived from MAC policy.

**HF12** A taint marking approach to confidentiality violation detection.

**NS05** Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software.

**HJR10** DIFC programs by automatic instrumentation.
Bibliography III

**CF07** Fine-Grained Information Flow Analysis and Enforcement in a Java Virtual Machine.

**Nal.07** A Virtual Machine Based Information Flow Control System for Policy Enforcement.

**asianhost.18** A MIPS-based coprocessor for information flow tracking in ARM SoCs.

**Heo.15** Implementing an Application-Specific Instruction-Set Processor for System-Level Dynamic Program Analysis Engines