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

Guillaume Hiet $\alpha$, Pascal Cotret $\beta$

$\alpha$ CentraleSupélec, guillaume.hiet@centralesupelec.fr
$\beta$ R&D engineer, pascal.cotret@gmail.com

June 22, 2018
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-STICCC/UBS @ Lorient
  - Guy Gogniat (Full Prof.), Vianney Lapôtre (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
Threat model

- Side-channel attacks not taken into account
- Software attacks: buffer overflow, ROP...
Dynamic Information Flow Tracking

**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 the flow of information with labeled containers and tagged files](image-url)
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
```

```
<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>
```

**pseudo-code**

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

<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](image)

<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>

<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>
Different Types of Information Flows

```python
S = secret_input()
A = S
print(A)
if A == "secret"
    B = False
    C = False
    B = True
    print(B)
    print(C)
    C = True
print(B)
print(C)
```
Different Types of Information Flows

```python
S = secret_input()
A = S
print(A)

if A == "secret"
    B = False
    C = False
    B = True
    print(B)
    print(C)
    C = True
    True
else
    False
    Executed branch

Direct Implicit flows

Indirect Implicit flows
```

Explicit flows

6/4 5
Different levels for DIFT

- **Fine-grained** (processor level)
  
  \[ \text{containers} = \text{addresses and registers} \]

- **Medium-grained** (language level)
  
  \[ \text{containers} = \text{variables} \]

- **Coarse-grained** (operating system level)
  
  \[ \text{containers} = \text{files, memory pages} \]
OS-level Software DIFC (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: Blare\textsuperscript{2}

Pros & Cons

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

\textsuperscript{1}Efstathopoulos et al. 2005; Zeldovich et al. 2006; Krohn et al. 2007.
\textsuperscript{2}Geller et al. 2011; Hauser et al. 2012.
Blare: Tainting Information at the OS Level

Information tags

- Contain **meta-information**, describing content
- Updated after each information flow to describe the new content
- They trace the origin of the content

![Diagram showing php.ini, index.php, /usr/bin/php, and site.db with annotations {1}, {2}, {}, and {}]

{9}
Information tags

- Contain **meta-information**, describing content
- Updated after each information flow to describe the new content
- They trace the origin of the content
Blare: Tainting Information at the OS Level

Information tags

- Contain **meta-information**, describing content
- Updated after each information flow to describe the new content
- They trace the origin of the content

![Diagram](image-url)
Information tags

- Contain **meta-information**, describing content
- Updated after each information flow to describe the new content
- They trace the origin of the content
Blare: Controlling Information Flows at the OS Level

Policy tags

- Contain **meta-information**, describing the legal content of the containers.
- It is set beforehand and doesn’t change.
- It is checked after each information flow and an alert is raised if the new content is not legal.

![Diagram](image.png)
Overapproximation Issue

KBlare considers processes as black boxes, so outputs are seen as a mix of all inputs. If we execute the code below:

```python
f1 = open('file1', 'r')
f2 = open('file2', 'w')
f2.write(f1.read())

f3 = open('file3', 'r')
f4 = open('file4', 'w')
f4.write(f3.read())
```

then:

\[
\text{tags}(f_2) = \text{tags}(f_1)
\]
\[
\text{tags}(f_4) = \text{tags}(f_1) \cup \text{tags}(f_3)
\]

This is obviously an overapproximation due to the black box approach.
**Application-level Software DIFC (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 Newsome and Song 2005; Harris, Jha, and Reps 2010.
4 Chandra and Franz 2007; Nair et al. 2007.
Hardware-based DIFT (fine-grained)

**Figure:** In-core DIFT \(^{5}\)

**Figure:** Dedicated CPU for DIFT \(^{6}\)

---


\(^{6}\) Nagarajan et al. 2008.
Hardware-based DIFT (fine-grained)

Figure: Dedicated DIFT co-processor

---

Fine-grained DIFT: comparison of the existing approaches

<table>
<thead>
<tr>
<th>HW-assisted</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 Few security policies</td>
</tr>
<tr>
<td>Dedicated CPU for DIFT</td>
<td>Low overhead (&lt;10%) Few modifications to CPU</td>
<td>Wasting resources Energy consumption (\times 2)</td>
</tr>
<tr>
<td>Dedicated DIFT coprocessor</td>
<td>Flexible security policies Low runtime overhead (&lt;10%) CPU not modified</td>
<td>Communication between CPU and DIFT Coprocessor</td>
</tr>
</tbody>
</table>
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
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
PTM Traces

CPU Cortex-A9

PTM

DIFT Monitor

Traces
(basic block addresses)

PFT Decoder

DIFT Core

System RAM
```c
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;
}
```

PTM trace: { ①; ②; ③; ⑤ }
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:

add r0, r1, r2 ⇒ r0 ← r1 ∪ r2
and r3, r4, r5 ⇒ r3 ← r4 ∪ r5
Static Analysis

- CPU Cortex-A9
- DIFT Monitor
- PTM
- User application
- Annotations
- Memory tags
- Source code
- Clang/LLVM
- Tag Register File
- PFT Decoder
- System RAM
- DIFT Core
- Traces (basic block addresses)
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 \texttt{r9} is dedicated for the instrumentation.
- The instrumentation FIFO address is retrieved via a UIO Driver.

Examples:

1. \texttt{ldr \ r0, [r2]} ⇒ \texttt{str \ r2, [r9]}
2. \texttt{str \ r3, [r4]} ⇒ \texttt{str \ r5, [r9]}

Instrumentation

**CPU Cortex-A9**

- PTM
  - Instrumentation (load/store addresses)
  - Traces (basic block addresses)

**DIFT Monitor**

- DIFT Core
- Instrumentation FIFO
- PFT Decoder
- Tag Register File

**User application**

- Source code
- Clang/LLVM

**Annotations**

**Memory tags**

**System RAM**
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.*
RfBlare: System calls

Linux Kernel with information flow support (RfBlare)

HardDrive (file system with extended attributes)

CPU Cortex-A9

PTM

System calls

Instrumentation (load/store addresses)

Traces (basic block addresses)

DIFT Monitor

Instrumentation FIFO

PFT Decoder

Tag Register File

User application

Annotations

Memory tags

System RAM

Source code

Clang/LLVM

Tag: passwd.txt

Tag: index.html

harddrive (file system with extended attributes)

ptm

system calls

cpu cortex-a9

instrumentation (load/store addresses)

traces (basic block addresses)

dift monitor

instrumentation fifo

pft decoder

tag register file

user application

annotations

memory tags

system ram

source code

clang/llvm
Overall architecture of the DIFT monitor

- **ARM Cortex-A9** connected to:
  - **PFT decoder** connected to **Decoded trace memory** connected to **DIFT Coprocessor**
  - **Interrupt**

- **DIFT Coprocessor** connected to:
  - **Tag memory (64 MB)**
  - **Tag annotations (64 MB)**
  - **DDR**

- **AXI interconnect** connected to:
  - **Instrumentation**
  - **Process mappings**
  - **RFBLare PS2PL**
  - **RFBLare PL2PS**

- **CPU memory (384 MB)**
What does a trace looks like?

**Code Source**

```c
int i;
for (i = 0; i < 10; i++)
```
What does a trace looks like?

**Code Source**

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

**Assembly**

```assembly
8638 for_loop:
...
b 8654:
...
866c: bcc 8654
```
What does a trace looks like?

**Code Source**

```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
21 2a 2a 2a 2a 2a 2a 2a 2a 2a
86 01 00 00 00 00 00 00 00 00
```
What does a trace looks like?

**Code Source**

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

**Assembly**

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

**Trace**

```
00 00 00 00 00 80 08 38 86 00 00
21 2a 2a 2a 2a 2a 2a 2a 2a 2a
86 01 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)
What does a trace looks like?

Decoded Trace
A-sync
Address 00008638, (I-sync Context 00000000, IB 21)
Address 00008654, Branch Address packet (× 10)
Dedicated DIFT coprocessor
Internal structure of the DIFT coprocessor
Extension for 2 threads - DIFT coprocessor

- **BRAM**
- **Annotations memory**
- **Decoded trace memory**
- **Annotations memory**
- **TMC (thread 1)**
- **TMC (thread 2)**
- **Dispatcher**
- **Tag memory**
- **Tag annotations**
- **Tag memory**
- **Tag memory**
- **Tag memory**
- **Tag memory**

- TagTRI T3,#4
- TagRR T1,T2
Extension for 2 threads - Trace details

Trace

<table>
<thead>
<tr>
<th>Decoded trace</th>
<th>Context ID</th>
<th>Stored address</th>
</tr>
</thead>
<tbody>
<tr>
<td>00010574</td>
<td>0004d2 42</td>
<td>00010574</td>
</tr>
<tr>
<td>00010428</td>
<td>0004d2 42</td>
<td>00010428</td>
</tr>
<tr>
<td>00010584</td>
<td>0004d2 42</td>
<td>00010584</td>
</tr>
<tr>
<td>000103c8</td>
<td>0004d2 42</td>
<td>000103c8</td>
</tr>
<tr>
<td>00010598</td>
<td>0004d2 42</td>
<td>00010598</td>
</tr>
<tr>
<td>000103f8</td>
<td>0004d2 42</td>
<td>000103f8</td>
</tr>
<tr>
<td>00010574</td>
<td>0004d3 42</td>
<td>00010574</td>
</tr>
<tr>
<td>00010428</td>
<td>0004d3 42</td>
<td>00010428</td>
</tr>
<tr>
<td>00010584</td>
<td>0004d3 42</td>
<td>00010584</td>
</tr>
<tr>
<td>00010575</td>
<td>0004d3 42</td>
<td>00010575</td>
</tr>
<tr>
<td>00010429</td>
<td>0004d3 42</td>
<td>00010429</td>
</tr>
<tr>
<td>00010585</td>
<td>0004d3 42</td>
<td>00010585</td>
</tr>
</tbody>
</table>
Overall architecture of the DIFT monitor

- **DIFT Coprocessor**
  - **Decoded trace memory**
  - **Tag annotations (64 MB)**
  - **Tag memory (64 MB)**

- **ARM Cortex-A9**
  - **PFT decoder**
  - **Interrupt**
  - **CPU memory (384 MB)**

- **AXI interconnect**
  - **Instrumentation**
  - **Process mappings**

- **Custom interface**
  - **RFBlare PS2PL**
  - **RFBlare PL2PS**

- **DDR AXI Master**

- **Developed**
  - **Adapted**
  - **Reused**

- **RFBlare Custom interface AXI interconnect**
### Area results (single-thread implementation)

<table>
<thead>
<tr>
<th>IP Name</th>
<th>Slice LUTs (in %)</th>
<th>Slice Registers (in %)</th>
<th>BRAM Tile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dispatcher</td>
<td>2223 (4.18%)</td>
<td>1867 (1.75%)</td>
<td>3</td>
</tr>
<tr>
<td>TMC</td>
<td>1837 (3.45%)</td>
<td>2581 (2.43%)</td>
<td>6</td>
</tr>
<tr>
<td>PFT Decoder</td>
<td>121 (0.23%)</td>
<td>231 (0.22%)</td>
<td>0</td>
</tr>
<tr>
<td>Instrumentation</td>
<td>676 (1.27%)</td>
<td>2108 (1.98%)</td>
<td>0</td>
</tr>
<tr>
<td>Blare PS2PL</td>
<td>662 (1.24%)</td>
<td>2106 (1.98%)</td>
<td>0</td>
</tr>
<tr>
<td>Blare PL2PS</td>
<td>62 (0.12%)</td>
<td>56 (0.05 %)</td>
<td>0</td>
</tr>
<tr>
<td>Decoded trace memory</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>AXI Master</td>
<td>858 (1.61%)</td>
<td>2223 (2.09 %)</td>
<td>0</td>
</tr>
<tr>
<td>TMMU</td>
<td>295 (0.55%)</td>
<td>112(0.10 %)</td>
<td>3</td>
</tr>
<tr>
<td>AXI Interconnect</td>
<td>2733 (5.14%)</td>
<td>2495 (2.34 %)</td>
<td>0</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>1381 (2.6%)</td>
<td>2160 (2.03%)</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total Design</strong></td>
<td><strong>10848 (20.39%)</strong></td>
<td><strong>15939 (14.98%)</strong></td>
<td><strong>14 (10%)</strong></td>
</tr>
<tr>
<td><strong>Total Available</strong></td>
<td><strong>53200</strong></td>
<td><strong>106400</strong></td>
<td><strong>140</strong></td>
</tr>
</tbody>
</table>
Memory footprint

![Bar chart showing normalized memory space overhead for different strategies and functions](image)

- **Original**
- **Strategy 1**
- **Strategy 2**

<table>
<thead>
<tr>
<th>Function</th>
<th>Original</th>
<th>Strategy 1</th>
<th>Strategy 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>choleski</td>
<td>1.11</td>
<td>1.13</td>
<td>1.15</td>
</tr>
<tr>
<td>crc</td>
<td>1.08</td>
<td>1.07</td>
<td>1.07</td>
</tr>
<tr>
<td>dft</td>
<td>1.07</td>
<td>1.10</td>
<td>1.12</td>
</tr>
<tr>
<td>fft</td>
<td>1.08</td>
<td>1.08</td>
<td>1.04</td>
</tr>
<tr>
<td>fir</td>
<td>1.10</td>
<td>1.10</td>
<td>1.03</td>
</tr>
<tr>
<td>lu</td>
<td>1.08</td>
<td>1.08</td>
<td>1.03</td>
</tr>
<tr>
<td>matrix</td>
<td>1.12</td>
<td>1.03</td>
<td>1.03</td>
</tr>
<tr>
<td>nbody</td>
<td>1.08</td>
<td>1.08</td>
<td>1.03</td>
</tr>
<tr>
<td>radix</td>
<td>1.03</td>
<td>1.03</td>
<td>1.03</td>
</tr>
<tr>
<td>wht</td>
<td>1.03</td>
<td>1.03</td>
<td>1.03</td>
</tr>
</tbody>
</table>
Some latency results

<table>
<thead>
<tr>
<th>Component</th>
<th>Original</th>
<th>Related work</th>
<th>Strategy 1</th>
<th>Strategy 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>choleski</td>
<td>9.56</td>
<td>2.84</td>
<td>1.67</td>
<td>1.50</td>
</tr>
<tr>
<td>crc</td>
<td>11.41</td>
<td>2.31</td>
<td>1.67</td>
<td>2.52</td>
</tr>
<tr>
<td>dft</td>
<td>3.64</td>
<td>1.67</td>
<td>3.73</td>
<td>3.73</td>
</tr>
<tr>
<td>fft</td>
<td>3.73</td>
<td>1.13</td>
<td>1.50</td>
<td>1.13</td>
</tr>
<tr>
<td>fir</td>
<td>2.84</td>
<td>2.52</td>
<td>1.67</td>
<td>2.92</td>
</tr>
<tr>
<td>lu</td>
<td>34.52</td>
<td>28.99</td>
<td>28.99</td>
<td>28.99</td>
</tr>
<tr>
<td>matrix</td>
<td>10.17</td>
<td>7.18</td>
<td>7.96</td>
<td>7.96</td>
</tr>
<tr>
<td>nbody</td>
<td>10.17</td>
<td>7.96</td>
<td>10.17</td>
<td>10.17</td>
</tr>
<tr>
<td>radix</td>
<td>11.60</td>
<td>2.92</td>
<td>11.60</td>
<td>11.60</td>
</tr>
</tbody>
</table>

Quick note

Enabling CoreSight components ⇒ Nearly no cost in terms of latency. Latency is only due to DIFT-related computations.
## Comparison with existing works

<table>
<thead>
<tr>
<th>Approaches</th>
<th>Kannan 8</th>
<th>Deng 9</th>
<th>Heo 10</th>
<th>Wahab 11</th>
<th>Latest work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardcore portability</td>
<td>No</td>
<td>No</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>
</tr>
<tr>
<td>Communication overhead</td>
<td>N/A</td>
<td>N/A</td>
<td>60%</td>
<td>5.4%</td>
<td>335%</td>
</tr>
<tr>
<td>Library instrumentation</td>
<td>N/A</td>
<td>N/A</td>
<td>partial</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>All information flows</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Area overhead</td>
<td>6.4%</td>
<td>14.8%</td>
<td>14.47%</td>
<td>0.47%</td>
<td>0.95%</td>
</tr>
<tr>
<td>Power overhead</td>
<td>N/A</td>
<td>6.3%</td>
<td>24%</td>
<td>16%</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>250 MHz</td>
</tr>
<tr>
<td>FP support</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>Yes</td>
</tr>
</tbody>
</table>

---

9Deng and Suh 2012.
10Heo et al. 2015.
Perspectives

Take away:
- CoreSight PTM allows to obtain runtime information (Program Flow)
- Non-intrusive tracing ⇒ Negligible performance overhead
- Support for multi-threaded and floating-point software
- Kernel support with RfBlare

Perspectives:
- Full PoC later this year (SoC files + Yocto)
- Intel / ST? (study)
- Multicore multi-thread IFT
HardBlare, a hardware/software co-design approach for Information Flow Control

Guillaume Hiet $^\alpha$, Pascal Cotret $^\beta$

$^\alpha$ CentraleSupélec, guillaume.hiet@centralesupelec.fr  
$^\beta$ R&D engineer, pascal.cotret@gmail.com

June 22, 2018

Many thanks to Muhammad, Mounir, Arnab, Vianney and Guy :)

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


Heo et al. 2015. Implementing an Application-Specific Instruction-Set Processor for System-Level Dynamic Program Analysis Engines.

Wahab et al. 2017. ARMHEx: A hardware extension for DIFT on ARM-based SoCs.
Efstathopoulos et al. 2005. Labels and event processes in the asbestos operating system.


Hauser et al. 2012. A taint marking approach to confidentiality violation detection.

Newsome and Song 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software.
Bibliography III

Harris, Jha, and Reps 2010. DIFC programs by automatic instrumentation.


Backup slides
TrustZone support

![Diagram showing TrustZone components and their interactions. The diagram includes CoreSight components, CPU, Memory (used by Linux OS), ARMHEx, and DIFT-related data. The connections indicate flow of execution trace, Ctrl signals, and instrumented data between trusted and untrusted areas.]