HardBlare: a Hardware-Assisted Approach for Dynamic Information Flow Tracking
Mounir Nasr Allah, Guillaume Hiet, Muhammad Abdul Wahab, Pascal Cotret, Guy Gogniat, Vianney Lapotre

To cite this version:

HAL Id: hal-01311032
https://centralesupelec.hal.science/hal-01311032
Submitted on 23 Jun 2016

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.
HardBlare proposes a software/hardware codesign methodology to ensure that security properties are preserved all along the execution of the system but also during files storage. The general context is to address Dynamic Information Flow Tracking (DIFT) that generally consists in attaching marks (also known as tags) to denote the type of information that are saved or generated within the system.

Let’s suppose that “print” function is public and the tag of a variable $x$ is underlined variable $\alpha$.

**Example code**

```plaintext
int p = 3;
int s = 42;
int x = p + s;
print(x);
```

<table>
<thead>
<tr>
<th>Example code</th>
<th>Tag initialization</th>
<th>Tag propagation</th>
<th>Tag check</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = p + s$;</td>
<td>$p \leftarrow $ public</td>
<td>$s \leftarrow secret$</td>
<td>if ($x != $ public) raise interruption</td>
</tr>
</tbody>
</table>

### Static Analysis

During the compilation phase, a static analysis is done on the LLVM intermediate representation produced from the source code, and propagated to the ARM backend for the machine code generation.

The result of static analysis gives a list of dependencies between information containers (e.g. registers, memory spaces...) for every basic blocks which are stored on a dedicated section in a ELF File.

During run-time, the Program Trace Macrocell (PTM) generates a trace containing the address for each committed instruction modifying the PC value.

Annotations related to the basic block identified by its address, given by the trace, are processed by the coprocessor to propagate tags.

#### Definitions

- **Tag dependencies** block contains annotations loaded when the program is launched.
- **Memory tags** block contains tags related to information containers.
- **Tag register file** contains tags related to CPU registers.

#### DIFT step-by-step

- **ARM CoreSight Components** export trace (for both CPUs) towards PL in PFT (Program Flow Trace) protocol.
- **PFT Decoder** decodes trace in usable format.
- Using decoded trace, **DIFT Coprocessor** reads tags dependencies block.
- **DIFT Coprocessor** looks for the tags either in memory or tag register file.
- **DIFT Coprocessor** computes tags depending on propagation rules.
- **DIFT Coprocessor** updates corresponding tags.
- **DIFT Coprocessor** checks for security policy violation and raise an interruption.

### State of the art

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software</td>
<td>Flexible security policies</td>
</tr>
<tr>
<td></td>
<td>Multiple attacks detected</td>
</tr>
<tr>
<td>Hardware</td>
<td>Low overhead (&lt;10%)</td>
</tr>
<tr>
<td></td>
<td>Invasive modifications</td>
</tr>
<tr>
<td>In-core DIFT</td>
<td>Low overhead (&lt;10%)</td>
</tr>
<tr>
<td></td>
<td>Few security policies</td>
</tr>
<tr>
<td>Dedicated CPU for DIFT</td>
<td>Low overhead (&lt;10%)</td>
</tr>
<tr>
<td></td>
<td>Few modifications to CPU Energy consumption ($x \times 2$)</td>
</tr>
<tr>
<td>Dedicated DIFT Coprocessor</td>
<td>Flexible security policies</td>
</tr>
<tr>
<td></td>
<td>Communication between CPU and DIFT Coprocessor</td>
</tr>
</tbody>
</table>

### References


