Monitoring program execution (and more!) on ARM processors
Hello!

- Embedded software security engineer
- Researcher in my spare-time
  (also former associate professor)

HardBlare project (3 labs, 2 PhDs...)
Threat model

Buffer overflow example with strcpy()

```c
void main()
{
    char source[] = "username2": // username2 to source[]
    char destination[7]: // Destination is 8 bytes
    strcpy(destination, source): // Copy source to destination
    return 0;
}
```

Playing with such attacks on ARM:

- [https://billy-ellis.github.io](https://billy-ellis.github.io) (@bellis1000)
- [https://www.root-me.org/?page=recherche&lang=en&recherche=ARM](https://www.root-me.org/?page=recherche&lang=en&recherche=ARM)
- [https://azeria-labs.com/](https://azeria-labs.com/) (@Fox0x01)
DIFT = Dynamic Information Flow Tracking

- DIFT => Detection of software attacks
  - Buffer overflow, Return Oriented Programming, etc.

- Security purposes => Integrity and Confidentiality

- Principle:
  - Tags attached to containers + relationship
  - At runtime, propagate tags
  - Detecting any violation at run-time asap
DIFT = Dynamic Information Flow Tracking

```
char buffer1[20], buffer2[20], buffer3[20];
FILE *password, *index, *unauthorized;

password = open("passwd.txt");
index = open("index.html");
unauthorized = open("unauthorized.html");

read(buffer1, password)
read(buffer2, index)
read(buffer3, unauthorized)

if(getuid()){
    send_to_socket(buffer2);
} else{
    send_to_socket(buffer3);
}
```
DIFT = Dynamic Information Flow Tracking

```c
char buffer1[20], buffer2[20], buffer3[20];
FILE *fpassword, *findex, *funauthorized;

fpassword = open("passwd.txt");
findex = open("index.html");
funauthorized = open("unauthorized.html");

read(buffer1, fpassword)
read(buffer2, findex)
read(buffer3, funauthorized)

if(getuid()){
    send_to_socket(buffer2);
} else{
    send_to_socket(buffer3);
}
```
DIFT = Dynamic Information Flow Tracking

```c
char buffer1[20], buffer2[20], buffer3[20];
FILE *password, *index, *unauthorized;

password = open("passwd.txt");
index = open("index.html");
authorized = open("authorized.html");

read(buffer1, password);
read(buffer2, index);
read(buffer3, unauthorized);
if(getuid()){
    send_to_socket(buffer2);
} else{
    send_to_socket(buffer3);
}
```
DIFT = Dynamic Information Flow Tracking

```c
char buffer1[20], buffer2[20], buffer3[20];
FILE *password, *index, *unauthorized;

password = open("passwd.txt");
index = open("index.html");
unauthorized = open("unauthorized.html");

read(buffer1, password)
read(buffer2, index)
read(buffer3, unauthorized)

if(getuid()){
    send_to_socket(buffer2);
} else{
    send_to_socket(buffer3);
}
```
Different levels for DIFT

- Operating system:
  Files / Executables

- Language level:
  Variables / Functions

- Processor level:
  Address, registers / Instructions
DIFT – Memory corruption detection

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>r1</td>
<td>&amp;input</td>
</tr>
<tr>
<td>r2</td>
<td>idx=input</td>
</tr>
<tr>
<td>r3</td>
<td>&amp;buffer</td>
</tr>
<tr>
<td>r4</td>
<td>&amp;buffer+idx</td>
</tr>
<tr>
<td>r5</td>
<td>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>
```

```
set r1 ← &tainted_input
load r2 ← M[r1]
add r4 ← r2 + r3
store M[r4] ← r5
```
DIFT – Memory corruption detection

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>r1:</td>
<td>&amp;input</td>
</tr>
<tr>
<td>r2:</td>
<td>idx=input</td>
</tr>
<tr>
<td>r3:</td>
<td>&amp;buffer</td>
</tr>
<tr>
<td>r4:</td>
<td>&amp;buffer+idx</td>
</tr>
<tr>
<td>r5:</td>
<td>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>
```

set r1 ← &tainted_input
load r2 ← M[r1]
add r4 ← r2 + r3
store M[r4] ← r5
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
```
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>r1</td>
<td>&amp;input</td>
</tr>
<tr>
<td>r2</td>
<td>idx = input</td>
</tr>
<tr>
<td>r3</td>
<td>&amp;buffer</td>
</tr>
<tr>
<td>r4</td>
<td>&amp;buffer+idx</td>
</tr>
<tr>
<td>r5</td>
<td>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>
```

Diagram:
- `set r1 ← &tainted_input`
- `load r2 ← M[r1]`
- `add r4 ← r2 + r3`
- `store M[r4] ← r5`
Attacker overwrites return address and takes control

```plaintext
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
```
Different levels for DIFT

- Tag initialization: data are tagged with theirs "security level"
  
  password="abcd" \( \text{Tag(password)} = \text{secret} \)

- Tag propagation: any new data derived from the tagged data is also tagged
  
  \( \text{log} = \text{err} + \text{password} \) \( \text{Tag(log)} = \text{Tag(err)} + \text{Tag(password)} \)

- Tag check: raise an exception if an information flow doesn’t respect a security policy
  
  \( \text{write(log, network)} \) Policy: (Tag(log) == public)
Different levels for DIFT

- **Application level**
  - Java / Android, Javascript, C

- **OS level**
  - kBlare (Linux kernel w/ software IFT)

- **Low level**
  - Deeping into processor architecture maybe?
Different levels for DIFT

- Application level
  - Java / Android, Javascript, C

- OS level
  - kBlare (Linux kernel w/ software IFT)

- Low level
  - Deeping into processor architecture maybe?

Buying an ARM license => no way. Or...
FPGA => Programmable electronics

Source: EEVBlog #496 – What is an FPGA? (Youtube)
Different levels for DIFT

In-core DIFT

Offloading
Different levels for DIFT

Off-core DIFT
## Related works

<table>
<thead>
<tr>
<th></th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software</td>
<td>Flexible security policies</td>
<td>Overhead (300% at least...)</td>
</tr>
<tr>
<td>In-core DIFT</td>
<td>Low overhead (10%)</td>
<td>Invasive modifications</td>
</tr>
<tr>
<td>Dedicated CPU</td>
<td>Low overhead (10%)</td>
<td>Wasting resources</td>
</tr>
<tr>
<td>Dedicated coprocessor</td>
<td>Low overhead (10%) CPU not modified</td>
<td>CPU/coprocessor communication</td>
</tr>
</tbody>
</table>
ARMHEx approach

- Limiting the impact of software instrumentation
- Security of the coprocessor
- First work on ARM-based SoCs
- Additional challenges
ARMHEX approach

- Limiting the impact of software instrumentation
- Security of the coprocessor
- First work on ARM-based SoCs
- Additional challenges
What can I do with my processor?
What can I do with my processor?

- CoreSight: debug components
- Available in most of Cortex-A + Cortex-M3 (for ARM)
- Can export stuff
CoreSight components – Where should I export my metadata?
Features:
- Trace filter
- Branch Broadcast
- Timestamping
- Etc, etc.
What does a trace look like?

**Source code**

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

**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 2a 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)
```
Our case:

- We want to store tags and initialize tags from the operating system:
  - Modified kBlare (based on a Linux Kernel 4.9)

- We don’t want to lose information (no over-approximation):
  - **Dynamic approach**: Instrumentation + PTM traces

- Extract some informations about the data flow (for tag propagation):
  - **Static Analysis**: Generating annotations.
Generating annotations

(status on late February)

- 200 instructions done:
  - LLVM meta-instructions
  - « Basic » stuff: add, compare, load/store, etc.

- TODO: 200 instructions left (at least...)
  - Parallel additions/subtractions features
  - Advanced SIMD instructions
DIFT toolchain

- Templates/tools/methods
- Custom embedded Linux
- HardBlare recipes added

Source: Bootlin (aka Free Electrons)
Coprocessor – Quick hints

- DIFT metadata protection
  - TrustZone + secure world
- Main challenge: speed!
Some latency results

![Graph showing latency results for various tasks with and without trace enabled.]
## Comparison w/ existing works

<table>
<thead>
<tr>
<th>Approaches</th>
<th>Kannan</th>
<th>Deng</th>
<th>Heo</th>
<th>ARMHEx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardcore portability</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Main CPU</td>
<td>Softcore</td>
<td>Softcore</td>
<td>Softcore</td>
<td><strong>Hardcore</strong></td>
</tr>
<tr>
<td>Communication overhead</td>
<td>N/A</td>
<td>N/A</td>
<td>60%</td>
<td>5.4%</td>
</tr>
<tr>
<td>Area overhead</td>
<td>6.4%</td>
<td>14.8%</td>
<td>14.47%</td>
<td>0.47%</td>
</tr>
<tr>
<td>Area (Gate Counts)</td>
<td>N/A</td>
<td>N/A</td>
<td>256177</td>
<td>128496</td>
</tr>
<tr>
<td>Power overhead</td>
<td>N/A</td>
<td>6.3%</td>
<td>24%</td>
<td>16%</td>
</tr>
<tr>
<td>Max frequency</td>
<td>N/A</td>
<td>256 MHz</td>
<td>N/A</td>
<td>250 MHz</td>
</tr>
<tr>
<td>Isolation</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td><strong>Yes</strong></td>
</tr>
</tbody>
</table>
Perspectives

Take away:
- CoreSight PTM allows to obtain runtime information (Program Flow)
- Non-intrusive tracing => Negligible performance overhead

RaspberryPi PoC (hopefully March)
Full PoC later this year (SoC files + Yocto)

Intel / ST? (study)

Multicore multi-thread IFT

Full-speed IFT
Monitoring program execution (and more!) on ARM processors

Pascal Cotret
pascal.cotret@gmail.com / @Pascal_r2

Many thanks to Muhammad, Mounir, Guy, Guillaume, Vianney and Arnab