Compiler-based Countermeasure Against Fault Attacks
Thierno Barry, Damien Couroussé, Bruno Robisson

To cite this version:
Thierno Barry, Damien Couroussé, Bruno Robisson. Compiler-based Countermeasure Against Fault Attacks. Workshop on Cryptographic Hardware and Embedded Systems, Sep 2015, Saint-Malo, France. . emse-01232664

HAL Id: emse-01232664
https://hal-emse.ccsd.cnrs.fr/emse-01232664
Submitted on 23 Nov 2015

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.
Compiler-based Countermeasure Against Fault Attacks

Thierno Barry*, Damien Courroussé*, Bruno Robisson**

*Univ. Grenoble Alpes, F-38000 Grenoble, France
**CEA-Tech DPACA, Gardanne, France
firstname.lastname@cea.fr

The goal is to implement the instruction duplication technique as a countermeasure against Fault Attacks on an ARM 32-bit Microcontroller[1,2]. Operating inside a compiler allowed us to reduce the security overhead thanks to the flexibility and code transformations opportunities offered by compilers.

The user identifies the portions of the program to protect

```c
@_to_secure_("fault")
int foo(int a, int b){
    ...
    return a * b + a;
}
```

The user has a full control over parts of the code to protect

Instructions cannot be duplicated at the middle-end due to the SSA form

<table>
<thead>
<tr>
<th>Entry</th>
<th>Attempted duplication</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%mul = mul %a, %b</code></td>
<td><code>%mul2 = mul %a, %b</code></td>
</tr>
<tr>
<td><code>%add = add %mul, %a</code></td>
<td><code>%add2 = add %mul, %a</code></td>
</tr>
</tbody>
</table>

We only select instructions that are suitable for duplication

1. Multiply and accumulate: `mla %a, %a, %b` is matched ✗
   - Instead we separately match: `mul` followed by `add` ✓

2. Generation of 3-address instructions:
   - Instead of generating `add vreg1, vreg2` ✗
   - We generate `add vreg3, vreg1, vreg2` ✓

Registers are allocated in favor of duplication

The register allocator tends to reduce register pressure: Reusing the allocated registers as soon as possible

When the liveness intervals (L) of registers are disjoint:

\[
\{L(vreg3)\} \cap \{L(vreg1) \cdot L(vreg2)\} = \emptyset
\]

We introduce a constraint: `$dst \neq $src$`

<table>
<thead>
<tr>
<th>Instruction Scheduling</th>
<th>Code Emission</th>
<th>Binary Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>register allocation</td>
<td>instruction selection</td>
<td>source code</td>
</tr>
<tr>
<td>instruction scheduling</td>
<td>code generation</td>
<td>LLVM bytecode</td>
</tr>
<tr>
<td>code emission</td>
<td>code elimination</td>
<td>target code</td>
</tr>
</tbody>
</table>

Comparison with assembly approach

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Transformation</th>
<th>Duplication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly approach</td>
<td><code>add r0, r0, r2</code></td>
<td><code>mov rx, r0, r0</code></td>
</tr>
<tr>
<td>Our approach</td>
<td><code>add r0, r1, r2</code></td>
<td><code>add r0, r1, r2</code></td>
</tr>
</tbody>
</table>

AES 8-bit NIST on ARM Cortex-M3

<table>
<thead>
<tr>
<th>Unprotected</th>
<th>Protected</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>6541 cycles</td>
<td>17311 cycles</td>
<td>× 2.63</td>
</tr>
</tbody>
</table>

Future Work & References

- Using code annotation for more flexibility when defining the code regions to protect
- Automatic identification of the most vulnerable parts of the program
- Compiler-based implementation of the masking countermeasure

References

[1] Barenghi et al. Countermeasures against fault attacks on software implemented AES

Legend

✓ Duplicable
✗ Not duplicable