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
#include <math.h>

int foo(int a, int b){
   ...
    return a + b + a;
}
```

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

### Instruction Selection

- Multiplying and accumulating: `mla` is matched

- We separately match: a `mul` followed by an `add` instead

### 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: \( dst \neq src \)

### Instructions are duplicated before scheduling

- Before duplication

- After scheduling

### 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, add r0, rx, r2</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>

**REFERENCES**

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


**LEGEND**

- ♦️ Duplicable
- ✗️ Not duplicable