Compiler-assisted security enhancement

Paolo Savini
Compiler Engineer Intern, Embecosm
Summary

• Information leakage
• LADA & SECURE
• Bit-slicing
• The ‘bit-slicer’
• A few considerations
Information leakage
Small electronic encrypting devices

Information leakage
Information leakage

Intrinsic features

- Power consumption
- Timing behaviour
- Electromagnetic leaks
- Sound emission
Information leakage
Side channel attacks

An attacker may use a side channel in order to gain sensitive information without the need of a flaw of the software or a brute force attack.
Side channel examples:

flow control

The pointer `p` points to sensitive data (the padding). According to the length of the padding the function returns. The execution flow depends on the length of the padding then

```c
p = buf;
switch( ctx->padding )
{
    case RSA_PKCS_V15:
        if( ++p != 0 )
            return( POLARSSL_ERR_RSA_INVALID_PADDING);
        bt = *++p;
        if( ( bt != RSA_CRYPT && mode == RSA_PRIVATE )
            || ( bt != RSA_SIGN && mode == RSA_PUBLIC ) )
        {
            return( POLARSSL_ERR_RSA_INVALID_PADDING);
        }
}
```

Security issue: CVE-2013-0169
Side channel examples:

**cache access**

The variable $y$ contains sensitive data from `buf` and is then used to access the array $R$. An attacker might gain some information about the content of $y$ by monitoring the cache events.

Security issue: CVE-2016-7440

```c
y = (int)(buf >> (DIGIT_BIT - 1)) & 1;
buf <<= (fp_digit) 1;

/*do ops*/
fp_mul(&R[0], &R[1], &R[y^1]);
fp_montgomery_reduce(&R[y^1], P, mp);

fp_sqr(&R[y], &R[y]);
fp_montgomery_reduce(&R[y], P, mp);
```
LADA & SECURE
The LADA project

Leakage Aware Design Automation

Development of tools that help the programmer design secure code and test it against leakage-related attacks
The LADA project

Leakage Aware Design Automation

Development of tools that help the programmer design secure code and test it against leakage-related attacks

Partnership with Embecosm → SECURE project
The SECURE project

Security Enhancing Compilation for use in Real Environments

Development and integration to the mainstream of LLVM and GCC of compiler tools that help the programmer write secure code
The SECURE project: aims

- Automatic selective bit-slicing
- Stack erasing
- Security warnings
Bit-slicing
Bit-slicing

Historically

Used in order to increase the word length of the processor before the advent of the microprocessor.
Bit-slicing:
Historically

Construction of a processor from modules of smaller bit width, such as an \texttt{n-bit processor} with \texttt{n 1-bit processors}

Software needed to be \texttt{properly} designed
Bit-slicing

In software

Software simulation of a parallel machine on a general purpose CPU

data slicing + instructions slicing
Bit-slicing: data slicing

Let's suppose that we need to bit-slice the array on the left, a new virtual register (that may be an element in a new array) is allocated to each of the bits of the original array.
Bit-slicing: instruction slicing

The algorithm used on bit-sliced data needs to be ‘bit-sliced’ as well: it must be turned into an equivalent algorithm made of atomic boolean operations, each addressing the proper slice of data.

```c
for ( i=0; i<n; i++ ){
    array3[i] = array1[i] ^ array2[i];
}
```

```c
for ( i=0; i<n; i++ ){
    for ( j=0; j<8; j++ ){
        array3_slices[i] = array1_slices[i] ^ array2_slices[i];
    }
}
```
Bit-slicing

What for?
Bit-slicing

What for?

Only some algorithms can be bit-sliced
Bit-slicing

What for?

Only some algorithms can be bit-sliced
And only some of these benefit from that (e.g. SIMD)
Given a SIMD system

<table>
<thead>
<tr>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
<th>B5</th>
<th>B6</th>
<th>B7</th>
</tr>
</thead>
<tbody>
<tr>
<td>10011001</td>
<td>10011001</td>
<td>10011001</td>
<td>10011001</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01001001</td>
<td>00100010</td>
<td>00100100</td>
<td>00101011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00010110</td>
<td>10100010</td>
<td>00000001</td>
<td>10011000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110000</td>
<td>11000100</td>
<td>00010010</td>
<td>00000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01001001</td>
<td>00001001</td>
<td>00000000</td>
<td>10010010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100100</td>
<td>00010010</td>
<td>00001001</td>
<td>10010011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10010010</td>
<td>11111111</td>
<td>00011101</td>
<td>10000010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10001110</td>
<td>01001010</td>
<td>11111110</td>
<td>00111101</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bit-slicing: What for?
In cryptography:

- Block ciphers are SIMD systems
- Input-independent execution time is key
In cryptography:

- Block ciphers are SIMD systems
- Input-independent execution time is key:
  - the execution time of boolean operations does not depend on the input
The ‘bit-slicer’
An LLVM pass that provides the programmer with:

- Automated bit-slicing of selected areas of the source code
- Simple data bit-slicing
Automated bit-slicing

```c
#pragma bitslice(array1, array2, array3)
{
   for ( i=0; i<n; i++ ){
      array3[i] = array1[i] ^ array2[i];
   }
}
```
The 'bit-slicer':
Automated bit-slicing

```c
#pragma bitslice(array1, array2, array3)
{
for ( i=0; i<n; i++ ){
    array3[i] = array1[i] ^ array2[i];
}
}

for ( i=0; i<n; i++ ){
    for ( j=0; j<8; j++ ){
        array3_slices[i] = array1_slices[i] ^ array2_slices[i];
    }
}
```
Simple data bit-slicing

The bit-slicer spares you from touching bit-sliced data

but what if we need to?
The 'bit-slicer':
Simple data bit-slicing

```c
#define N 10

uint8_t array[N];
slice_t array_slices[BLOCK_LEN * 8];

__builtin_get_bitsliced_data(array, array_slices);
```
A few considerations
A few considerations

Bit-slicing is very niche technique

Never forget the cons:

- Increase of allocated space
- Increase of code size (to create and manage the slices)
- Only some algorithms can be efficiently bit-sliced
SIMD programs are the best candidates:
increased throughput

A few considerations
SIMD programs are the best candidates:
- increased throughput

Block ciphers:
- increased throughput
- input independent execution time
SIMD programs are the best candidates:
increased throughput

Block ciphers:
increased throughput
input independent execution time
→ resistance against timing side channel attacks
A few considerations

BUT!
A few considerations

Any operation that involves a dependency among the bits of an operand (like the carry in an addition) might cause a loss of efficiency or may not even be bit-sliced at all.
A few considerations

Even among block ciphers it might well be that only some implementations of the same block cipher benefit from bit-slicing.
Questions?

• Information leakage
• LADA & SECURE
• Bit-slicing
• The ‘bit-slicer’
• A few considerations
Examples of bit-sliced implementations

Faster and Timing-Attack Resistant AES-GCM
https://link.springer.com/chapter/10.1007/978-3-642-04138-9_1

Lightweight Fault Attack Resistance in Software Using Intra-Instruction Redundancy
https://eprint.iacr.org/2016/850
Contact

Email: paolo.savini@embecosm.com

Linkedin: www.linkedin.com/in/paolo-savini-56b833147