Analysis of Assembly Execution Time

Nicolas Boichat, Eric Bisolfati

May 28, 2009

Outline

Assembly execution time

Assume this trivial snippet of code:

```
int mult(int a, int b) {
    int i, sum = 0;
    for (i = 0; i < b; i++)
      sum += a;
    return sum;
}
```

In this example:

- The **time** spent in the for loop **depends** on `b`.
- If it is a cryptographic function → **time will leak information**.

Leak detection

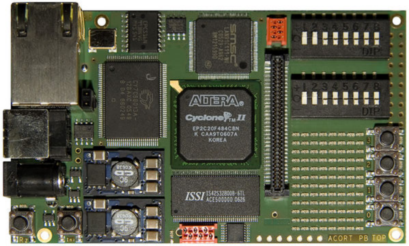
Can we find a way to **detect these leaks**?

We analyse **assembly code**, it has 2 main advantages:

1. Execution time depends on the **assembly code** only.
2. We know **how much time each instruction lasts**.

Simplifying the problem

- x86 assembly is complex.
- We used a **softcore CPU** on the FPGA4U: `Nios II/e`.
- **Simple architecture**, similar to microcontrollers:
  - No cache, no pipelining.
  - **Deterministic execution time**.



Intruction set

- **Nicely encoded instruction set**.
- First step $\longrightarrow$ **write a disassembler**.

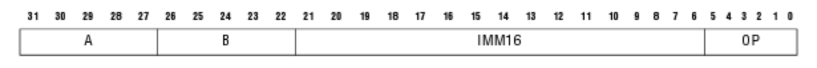


Figure: Format of an I-type instruction.

Control flow graph

From the disassembled code, we can generate a **control flow graph**.

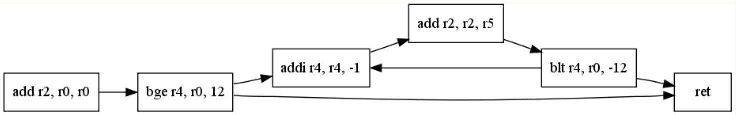


Figure: A control flow graph.

Data dependencies

We keep a map for each instruction, indicating **register dependencies**.

```
x   : add r2, r4, r5 -- deps: r2 -> (r4,r5)
x+4 : add r3, r2, r6 -- deps: r2 -> (r4,r5);
                              r3 -> (r4,r5,r6)
```

The dependencies are **forwarded** in the control graph.

Worklist algorithm

We start with a **worklist** with the **first node** in it.

Then we **repeat** the following:

1. **Pick an element** of the worklist.
2. **Merge** dependencies from **predecessors**.
3. **Analyse** current instruction.
4. If **dependencies have changed** → **add successors** to the worklist.

Convergence

We are working in a **lattice**:

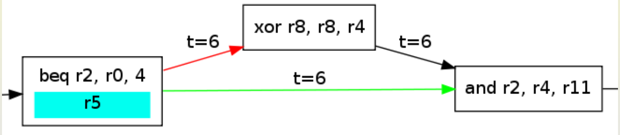
- Dependencies are a **list of sets**.
- Easy to define a **partial order** on it.

Our update function is **monotonic** (we only add dependencies, never remove).

→ **Tarski's fixed point theorem** guarantees that we will **converge** eventually.

Control and time dependencies

Consider the following example (r2 depends on r5):



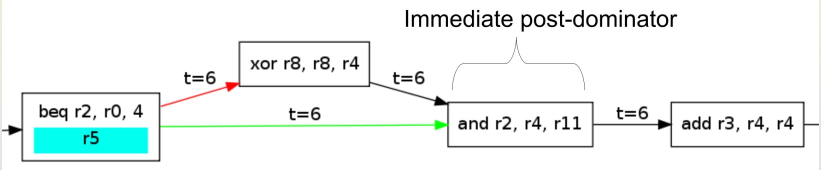
→ The **execution time** will depend on r5.
→ The **value** of r8 will depend on r5.

Time and control dependencies

To find whether a branch introduces **time or control dependency**, we need to find the **merging point**.

Post dominators

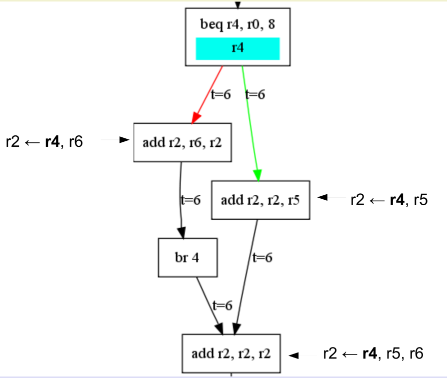
Find the **immediate post-dominator** of the branch instruction, i.e. the first instruction that will be executed whether the condition is **true** or **false** (→ the merging point).



Well-known algorithms exist to compute those (ex: Tarjan's algorithm).

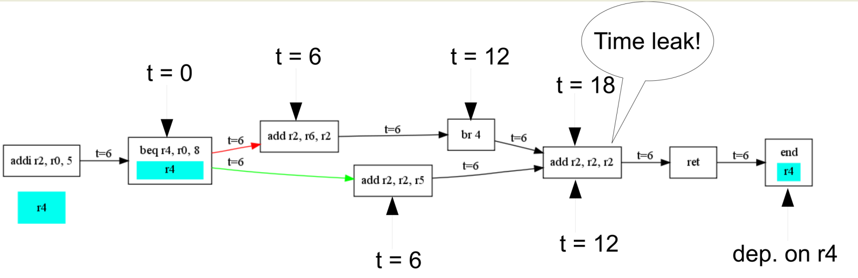
Control flow dependencies

All operations **results** are **tainted** by the **branch condition**, until we reach a post-dominator.



Time dependency

- **Time** is **recorded** in **both branches**.
- If 2 branches merge with a **different time**, we add a **time dependency** on the branch condition.



Function calls

- **Function calls** are handled
- We need to manage the **stack** as well
- Very simple **symbolic execution** to know the stack pointer.

Study case - Multiplication in Gallois field

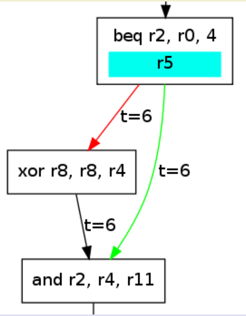
Multiplication in Gallois field:

```
 /* Multiply two numbers in the GF(2^8) finite field defined
  * by the polynomial x^8 + x^4 + x^3 + x + 1 */
uint8_t gmul(uint8_t a, uint8_t b) {
  uint8_t p = 0;
  uint8_t counter;
  uint8_t hi_bit_set;
  for(counter = 0; counter < 8; counter++) {
    if((b & 1) == 1)
      p ^= a;
    hi_bit_set = (a & 0x80);
    a <<= 1;
    if(hi_bit_set == 0x80)
      a ^= 0x1b; /* x^8 + x^4 + x^3 + x + 1 */
    b >>= 1;
  }
  return p;
}
```

Unsafe version

Now, consider b contains some **sensitive information** (e.g. the key being used).

```
for(counter=0; counter < 8; counter++) {
  if((b & 1) == 1)
    p ^= a;
  ...
}
```



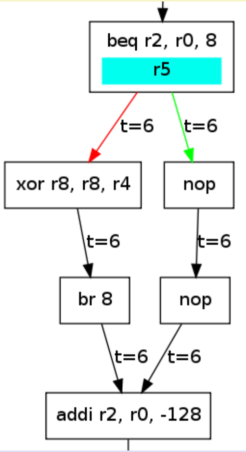
Our tool reports a **time dependency** on b.

Safe version

Let's fix that by adding some nops:

```
for(counter=0; counter < 8; counter++) {
  if((b & 1) == 1) {
    p ^= a;
  } else {
    asm volatile( "nop" );
    asm volatile( "nop" );
  }
  ...
}
```

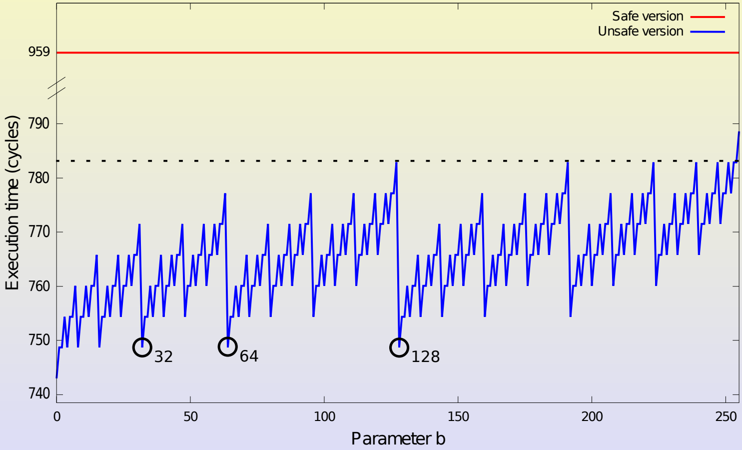
Our tool reports **no time dependency** on b.



Experimental verification

- We can **test both versions** on an FPGA4U.
- **Measure the execution time** as a function of `b`.

Experimental verification



Experimental verification

In this case, **our tool provides correct results**.

Future work

- Analyse **memory operations**.
- Handle **recursive function calls**.
- Analysis of **more complex programs**.
- Analysis of **pipelined processors**.

Thanks for your attention.

Questions ?

Backup

Backup

Lattice definition

Registers:

$$S = \{\texttt{r1}, \ldots \texttt{r31}\}$$

Dependencies:

$$(d_{\texttt{r1}}, d_{\texttt{r2}}, \ldots d_{\texttt{r31}}),\ d_* \in 2^S$$

Lattice:

$$(d_{\texttt{r1}}, \ldots d_{\texttt{r31}}) \sqsubseteq (d'_{\texttt{r1}}, \ldots d'_{\texttt{r31}}) \leftrightarrow \bigwedge_{i \in S} d_i \subseteq d'_i$$