Countermeasures for the Simple Branch Prediction Analysis

Giovanni Agosta and Gerardo Pelosi
Dipartimento di Elettronica e Informazione
Politecnico di Milano
{agosta,pelosi}@elet.polimi.it

December 21, 2006

Abstract

Branch Prediction Analysis has been proposed as an attack method to obtain key bits from a cryptographic application. In this report, we put forth several solutions to avoid or prevent this attack. The reported countermeasures require only minimal hardware support that is commonly available in modern superscalar processors.

1 Introduction

In [1, 2] Acıiçmez *et al.* propose an attack method that exploits the Branch Target Buffer (BTB) [3] as a log of the branching choices performed by a cryptographic primitive. Basically, the idea is that, by performing the same branch repeatedly, a *spy* process can force the cryptographic process to always have mispredicted branches. Therefore, the cryptographic process causes the BTB to be modified when the attacked branch is taken, and left to the attacker's branch target address when the attacked branch is not taken. This attack, being based on a log of the branching choices that is visible to all processes, can allow an unprivileged spy process to quickly infer the key used by the cryptographic process, since the attacker can read this log by simply measuring the time needed to perform its own branches: longer times correspond to mispredicted branches, i.e. to branches taken in the cryptographic process, while shorter times correspond to not taken branches. With respect to other timing attacks, this technique is immune to countermeasures such as branch balancing and blinding, since it does not measure computation times in the attacked process.

2 Proposed Countermeasures

Several simple hardware and/or software techniques can actually be employed to block the attack described in [1, 2]. Obviously, a simple and effective solution would be to allow sensitive processes to disable the access to the BTB unit. This, however, requires a new generation of processors, which, for general purpose chips is probably too long a time. Therefore, we propose a set of software and compiler techniques to address the vulnerability while no appropriate hardware countermeasure is available.

First of all, most modern processors have predicated instructions. These can be used to remove some sensitive branches, by converting them into instructions belonging to a single control flow. For example, the following code:

$$if(a)\{b = c + d;\}$$

could be replaced by:

```
cmpi r1, r2, 0
add r3, r4, r5
select r2, r3, r1
```

where the `select` operation assigns to the destination register $r2$ the value of $r3$ if $r1$ is not zero. For the other instructions, the first operand is always the destination. Modern instruction sets such as the ARM and IA64 are fully predicated, so there would be no need for an explicit conditional assignement, so that only two instructions would be needed.

This solution is attractive, because it does not affect the performances in processors that exhibit a sufficient degree of instruction-level parallelism (ILP) – actually, it can even improve the performances. However, it can only be applied to a reduced number of branches, since it can cause a significant growth of the code issued in each execution path.

If predicated instructions are not available in the target ISA, the code can still be rewritten to avoid branching, at least when the bodies of the then and else parts of the branch are small enough. An example of such technique is as follows:

$$if(a)\{b = c \oplus d;\}$$

becomes:

$$a_{then} = (a! = 0) \times 0\text{xffffffff};$$

$$a_{else} = (a == 0) \times 0\text{xffffffff};$$

$$x = c \oplus d;$$

$$b = b \,\&\, a_{else} + x \,\&\, a_{then};$$

Sometimes, however, even this technique can be impractical, since the bodies of the then and else parts may be large enough, or contain instructions with side effects (e.g., stores in memory). In this case, a third technique can be employed to ensure that the side channel attack on the BTB fails. The BTB attack is

based on the fact that there is a *conditional* branch in the code – therefore, an effective way to block it is to remove all conditional branches from the sensitive code, and replace them with indirect branches, as in the following example:

```
bz r1, label
   <then part>
jmp end
label:
   <else part>
end:
```

where $r1$ contains the result of the condition expression (let us assume it can only be 0 or 1). The branch instruction is replaced by the following code:

```
add r2, r3, r1
ld r4, 0(r2)
jmpl r4
```

where the addresses of the then and else code fragments are stored in memory at locations pointed by $r3$ and $r3+1$, and `jmpl` is an indirect branch reading the address from a register. The new code snippet loads the target address from the correct position and *always* perform an indirect branch, regardless of whether the condition is true or false – there is no fallthrough between contiguous basic blocks. The attacker process makes it so that the branch is always mispredicted, so it will always find its own branches to be mispredicted as well – the BTB will not contain useful information anymore.

This last technique can always be applied, and it can be applied by means of a simple compiler pass that replaces the generation of direct branches with appropriate indirect branches, at a minimal added cost (one load, one add and the jump will always be taken). Moreover, the same technique could be applied on existing compiled code, since it works directly on the assembler code – when the position of the basic blocks in memory is already known. Indirect branches are available in most architectures, including x86, IA64, MIPS and ARM, making the technique widely applicable.

The attack in [1, 2] was proposed as an attack against the implementations of the RSA cryptosystem, using the OpenSSL implementation as a test case. In the OpenSSL case, it would be possible to use the predication or if-elimination techniques, but other implementations or other cryptosystems might still be vulnerable. Moreover, in closed source cryptosystems, it is impossible to ascertain that a suitable design has been employed – an implementation of our last proposed technique as a dynamic or link-time optimization may still be used to make the code safe.

3 Concluding Remarks

In this report, we put forth several solutions to avoid or prevent the side channel attack on the BTB proposed in [1, 2]. The reported countermeasures can be

easily implemented by retargeting the compiler, once the target architecture is known, or as a link-time or dynamic code optimization.

References

[1] Onur Aciicmez, Cetin Kaya Koc, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. Cryptology ePrint Archive, Report 2006/351, 2006. `http://eprint.iacr.org/`.

[2] Onur Aciicmez, Jean-Pierre Seifert, and Cetin Kaya Koc. Predicting secret keys via branch prediction. Cryptology ePrint Archive, Report 2006/288, 2006. `http://eprint.iacr.org/`.

[3] Bradley D. Hoyt, Glenn J. Hinton, Andrew F. Glew, and Subramanian Natarajan. Branch target buffer for dynamically predicting branch instruction outcomes using a predicted branch history. US Patent 08/509331, Internation Class G06F 9/38, 1996.