Countermeasures Against Branch Target Buffer Attacks*

Giovanni Agosta, Luca Breveglieri
Dipartimento di Elettronica e Informazione, Politecnico di Milano,
Piazza L. Da Vinci 32, 20133 Milano, Italy
{agosta,brevegli}@elet.polimi.it

Gerardo Pelosi
Dipartimento di Ingegneria dell'Informazione e Metodi Matematici,
Università degli Studi di Bergamo,
Viale Marconi 5, 20044 Dalmine (BG), Italy
gerardo.pelosi@unibg.it

Israel Koren
Department of Electrical & Computer Engineering, University of Massachusetts,
Amherst, MA 01003
koren@ecs.umass.edu

Abstract

Branch Prediction Analysis has been recently proposed as an attack method to extract the key from software implementations of the RSA public key cryptographic algorithm. In this paper, we describe several solutions to protect against such an attack and analyze their impact on the execution time of the cryptographic algorithm. We show that the code transformations required for protection against branch target buffer attacks can be automated and impose only a negligible performance penalty.

1 Introduction

Timing attacks against secret-key/public-key cryptosystems attempt to recover complete key information by measuring the running time of certain computations during the encryption or decryption processes. Timing attacks have been developed against many common crypto-algorithms (e.g., RSA, DSA, Diffie-Hellman and RC5) relying on the key-dependent correlation between the input data and the execution time [7]. The reasons for the dependence of the execution time on the key include conditional branches, cache hits/misses and processor instructions executed in a non-fixed time.

For a timing attack to succeed, it must be possible to measure the running time of cryptographic operations. Such measurements can be done not only on a smart card which is in the attacker's possession, but also on a software implementation of a crypto-algorithm running on a remote machine [1].

Several techniques have been developed to protect cryptographic systems against timing attacks, e.g., by forcing the execution time to be independent of the key bits.

In [5, 6] Acıiçmez *et al.* propose a new attack method that exploits the fact that the Branch Target Buffer (BTB) [3] keeps a history log of the branching choices performed by a cryptographic primitive. The attacker will run a *spy* process on the same multi-threaded processor that is executing the cryptographic process and will take advantage of the fact that both processes share the use of the BTB. The basic idea is that the spy process will execute a sufficiently high number of branches to guarantee that the BTB entries that keep track of the branches executed by the cryptographic process will be replaced, thus forcing the cryptographic process to always have mispredicted branches. Then, when the cryptographic process executes, it will cause the BTB to be modified when the attacked branch is taken, and leave the spy process' branch target address intact in the BTB when the attacked branch is not taken. This attack, being based on a log of the branching choices that is common to all processes, enables an unprivileged spy process to quickly infer the key used by the

*This work was carried out under partial financial support of the Italian MiUR (Project PRIN 2006 ID-2006099978) and in part by project FSE ID-413174.

cryptographic process, since the attacker can reconstruct the 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. Compared to the classical timing attacks, this technique is immune to countermeasures such as branch balancing and blinding, since it does not measure computation time in the attacked process.

2 Countermeasures

Several simple hardware and/or software techniques can be employed to protect against the attack described in [5, 6]. Obviously, a simple and effective solution would be to allow sensitive processes to disable the access to the BTB unit. This, however, requires a modification to current processor designs; a modification that is unlikely to be implemented in general-purpose microprocessors in the near future. Other solutions have been proposed in a wider framework of control-flow side channel attacks, which can be effectively employed to face an attack to the BTB. These solutions rely on eliminating conditional branches and are described next.

2.1 Branch Elimination

Branch elimination techniques are based on the fact that, when there are no control operations (branches or a loop with a variable number of iterations), then obviously no control flow attack can be mounted [9]. It is quite straightforward to see that BTB attacks are a special case of control flow attack, since they rely on the BTB to convey information about the actual path in the cryptographic code followed during its execution. Therefore, the code can 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 a technique is as follows:

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

becomes:

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

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

$$x = c \oplus d;$$

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

Molnar *et al.* [9] show how to transform a program in order to obtain a *program counter-secure* (*PC-secure*) version of it. Basically, the technique ensures that all the conditional branches are removed, so that every execution of the program has exactly the same sequence of program counter values. This technique is extremely effective, as it not only protects against control flow attacks (and consequently against timing attacks), but also against attacks that exploit the knowledge of the data accesses. The latter is true since the data memory addresses accessed by the transformed program are independent of the control flow of the original program, as it happens, for example, for Coron's exponentiation method [2], where a conditional assignment originally expressed as:

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

becomes

$$tmp[1] = c \oplus d;$$

$$tmp[0] = b;$$

$$b = tmp[a];$$

where a is a Boolean value. Such a solution, while PC-secure, is unsecure with respect to attacks that exploit the knowledge of an accessed memory address, since the attacker can infer the value of a from the address of the datum read in the last instruction. As a variant of the technique in [9], *predicated execution* (also called conditional instruction) [4, 8] can be employed to the same effect. Since most modern processors have predicated (conditional) instructions, these can be used to remove sensitive branches by converting them into instructions belonging to a single control flow. For example, the following code fragment:

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

could be replaced by:

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

$$if\ (r2 == 0)\ \{\, r1 = 1\,\}$$
$$else\ \ \{\, r1 = 0\,\}$$
$$r3 = r4 + r5$$
$$if\ (r1! = 0)\ \{\, r2 = r3\,\}$$

where the `select` operation assigns 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 that there would not be a need for an explicit conditional assignment, and only two instructions would be required.

Molnar's solution [9] is attractive, because it does not affect the performance of processors which exhibit a sufficient degree of instruction-level parallelism (ILP). Actually, the use of predicated execution may even improve the performance. Sometimes, however, this technique can become impractical when the bodies of the *then* and *else* parts are very large, or contain function calls, so that inlining is required to make sure that the function parameters that affect

the length of loops are known at compile time – a mandatory condition [9] to guarantee PC-security. These lead to a significant increase in the code size and more importantly, in execution time. Case studies in [9] report slowdown factors of up to 5 times as well as a stack size increase of up to 2 times. Moreover, some loops may be driven by values known only at runtime (e.g., input values), thus making a PC-secure version of the program impossible to obtain.

2.2 Branch to Indirect Jump Conversion

A different technique can be employed to ensure that the side channel attack on the BTB will fail. The BTB attack is based on the fact that there is a *conditional* branch in the code, therefore, an effective way to protect against it is to remove all the conditional branches from the sensitive code, and replace them with indirect branches, as shown 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
```

$r2 = r3 + r1$
$r4 = Mem(0 + r2)$
$PC = r4$

where the addresses of the *then* and *else* code fragments are stored in the memory locations pointed at 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* performs an indirect branch, regardless of whether the condition is true or false – there is no fall through between contiguous basic blocks. The attacker process causes the branch to be always mispredicted, so that it will always find its own branches to be mispredicted as well, and the BTB will no more contain useful information. The same technique can be applied to the source code as well, by replacing an *if-then-else* conditional statement with:

```
JUMP(cond,lthen,lelse);
lthen :
   ...
lelse :
   ...
ljoin :
   ...
```

where the `JUMP` macro is defined by the following code:

```
__inline__ unsigned int cneg(unsigned int x) {
   unsigned int y;
   __asm__  ("movl\t$0, %0\n\t"
             "cmpl\t$0, %1\n\t"
             "movl\t$0xFFFFFFFF, %%ecx\n\t"
                "cmovnzl\t%%ecx, %0\n"
                   : "=d"(y)
                   : "m" (x)
                   : "ecx" );
   return(y);
} /* end cneg */
#define SELECT(x,v1,v2) \
   ( (cneg(x)&v1) | ( (~(cneg(x)))&v2) )
#define JUMP(x,lthen,lelse) goto *(void *) \
  SELECT(x, (unsigned int)(&&lthen), \
          (unsigned int)(&&lelse))
```

This last technique is always applicable and can be implemented by means of a simple compiler pass that replaces direct branches with appropriate indirect branches, at a minimal overhead (one load, one add and a branch that will always be taken). The technique can be applied at the source code level as well, using a trivial transformation (implemented by a parser) that leaves all the code unchanged except for the generation of the `JUMP` macro and related targets according to the conditional statement rules of the language grammar.

Moreover, the same technique can be applied to existing compiled code, as it works directly on the binary code; when the position of the basic blocks in the memory is already known. Therefore, the technique is suitable for implementation in link-time or dynamic optimizers.

Indirect branches are available in most architectures, including x86, IA64, MIPS and ARM, which makes the technique widely applicable to commercial platforms. While this technique does not qualify as PC-secure (each branch of a conditional is still executed on a different set of program counter values), it is capable of countering attacks to the BTB with negiligble performance impact. Hence, the technique fits well applications in a real world context.

The attack in [5, 6] was mounted against implementations of the RSA cryptosystem, using the OpenSSL implementation as a test case. In the OpenSSL case, it would be possible to use the predicate execution or the if-elimination techniques, but different implementations or other cryptosystems might still be vulnerable. Moreover, in closed source cryptosystems it is impossible to ascertain whether a suitable design is employed, therefore an implementation of our proposed indirect branch technique as a dynamic or link-time optimization can still be used to secure the code.

3 Performance Evaluation

To compare the effectiveness of the discussed countermeasures, we applied them to an RSA square-and-multiply algorithm implemented using the OpenSSL library, which has been subjected to the attack presented in [5, 6]. The test

Table 1. Profiling of the proposed methods. The leftmost three columns provide measures relative to individual branches, the rightmost gives the execution time for the RSA square-and-multiply for each technique.

Implementation method	Branch penalty	Footprint penalty	Data reference penalty	Time [clk] (1024-bit key)
Original code	1.00	1.0	0	59,698
Coron's method	1.71	0.8	2	58,756
Predicated conditional	4.79	1.2	4	58,967
Indirect jump (**our**)	**4.83**	**2.0**	**3**	**61,846**

case was built using the same simplifications as in [5, 6], and the measures were collected using the Valgrind profiling tools suite [10]. Table 1 compares the previously discussed three methods to protect against the simple BTB attack. The first column in the table shows the execution time penalty (expressed as a slowdown factor) for a single high-level conditional construct: both the indirect jump conversion and the predicate version of the conditional construct impose, for different reasons, a relatively high slowdown. In fact, each conditional executes almost five times slower than the original. Coron's method, on the other hand, only incurs a limited penalty.

The second column in the table shows the increase in the static footprint of each conditional branch relative to the original size; that is, the number of machine instructions used to represent it in the object code. In this case, Coron's method allows a more compact representation than the original code, while the other two methods incur small penalties. The third column reports the number of additional data references that are performed by the transformed code for each conditional branch. Each of the three methods imposes a different penalty, with the predicated conditional incurring the highest penalty: it adds four data references for each conditional branch. This metric is especially relevant since a large number of additional memory references can negatively impact the data cache miss rate, and therefore increase the average delay of a read or write operation in the memory.

The first three columns in Table 1 measure the impact of the transformation used to secure a small code section (a single conditional branch) against simple BTB attacks. The last column of Table 1, on the other hand, shows the overall impact of these transformations on the computation time of the core routine of the RSA encryption algorithm, thus giving a better estimate of the performance impact. We can see that the execution time penalties are very small since the transformations are applied only to a limited portion of the code where sensitive information is used to compute a branch outcome.

Finally, Figure 1 shows the memory usage profile of the RSA square-and-multiply algorithm. Since the stack memory usage is not significantly affected by the countermeasures, the memory usage profiles of the original version and all three modified versions (with the countermeasures) are almost identical and thus, we show only the graph for the indirect jump version. This result again demonstrates the minimal impact of the various countermeasures on the execution of the cryptographic algorithm.

The worst case conditions exposed by the first three columns in Table 1 are still useful when attemping to gauge the impact of the countermeasures, when applied to, for example, legacy binary code that is not secure. In such a case, our proposed indirect jump technique is the only one that can be easily implemented.

4 Concluding Remarks

In this paper, we evaluated the currently existing solutions and compared them to a new technique which we propose for protecting against the BTB side channel attack introduced in [5, 6].

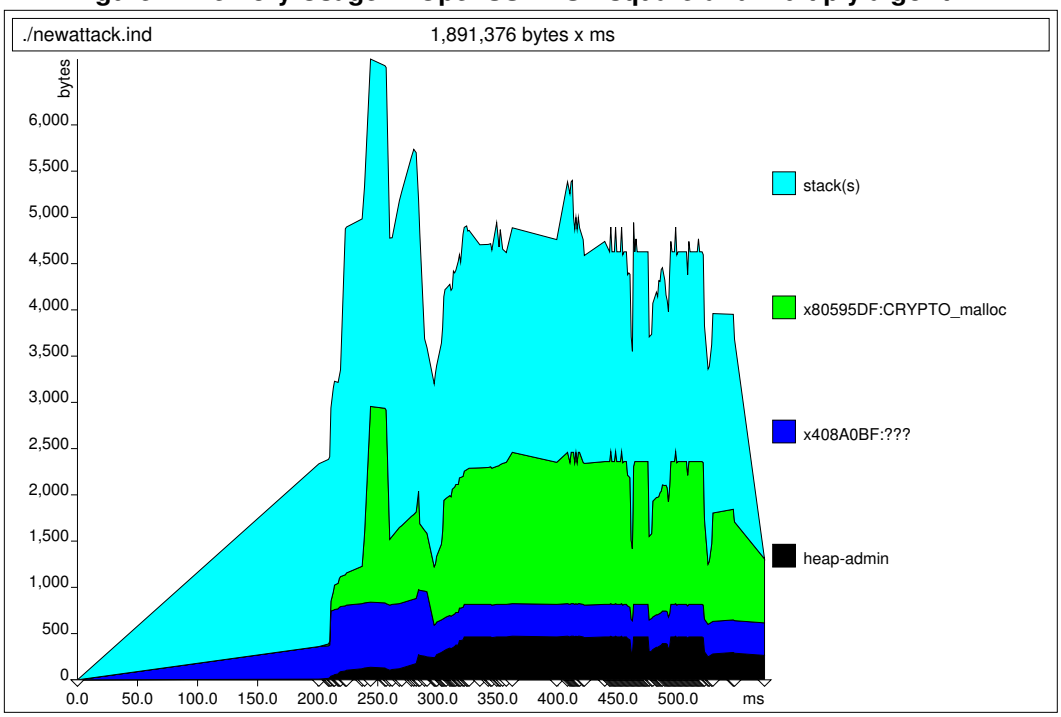
The countermeasures can be easily implemented either in high-level code, by a simple source code transformation, or by re-targeting the compiler, once the target architecture is known, or even through link-time or dynamic code optimization.

References

[1] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[2] Jean-Sébastien Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In *CHES '99: Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems*, pages 292–302, London, UK, 1999. Springer-Verlag.

Figure 1. Memory Usage in OpenSSL RSA square-and-multiply algorithm



[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.

[4] P. Y. T. Hsu and E. S. Davidson. Highly concurrent scalar processing. In *ISCA '86: Proceedings of the 13th annual international symposium on Computer architecture*, pages 386–395, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[5] Onur Acıiçmez, Çetín Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *ACM Symposium on Information, Computer and Communications Security, ASIACCS 2007*, Mar 2007.

[6] Onur Acıiçmez, Jean-Pierre Seifert, and Çetín Kaya Koç. Predicting secret keys via branch prediction. In *Topics in Cryptology, The Cryptographers' Track at the RSA Conference, CT-RSA 2007*, volume 4377 of *LNCS*, pages 225–242. Springer, Feb 2007.

[7] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *CRYPTO*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[8] Scott A. Mahlke, Richard E. Hank, James E. McCormick, David I. August, and Wen-Mei W. Hwu. A comparison of full and partial predicated execution support for ilp processors. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 138–150, New York, NY, USA, 1995. ACM Press.

[9] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In Dongho Won and Seungjoo Kim, editors, *ICISC*, volume 3935 of *Lecture Notes in Computer Science*, pages 156–168. Springer, 2005.

[10] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electr. Notes Theor. Comput. Sci.*, 89(2), 2003.