A Note On Side-Channels Resulting From Dynamic Compilation

D. Page

University of Bristol,
Merchant Venturers Building,
Woodland Road,
Bristol, BS8 1UB, UK.
page@cs.bris.ac.uk

Abstract. Dynamic compilation systems are of fundamental importance to high performance execution of interpreted languages such as Java. These systems analyse the performance of an application at run-time and aggressively re-compile and optimise code which is deemed critical to performance. However, the premise that the code executed is not the same code as written by the programmer raises a number of important security concerns. In this paper we examine the specific problem that dynamic compilation, through transformation of the code, may introduce side-channel vulnerabilities where before there were none.

1 Introduction

Dynamic compilation techniques have been a major factor in allowing interpreted languages, such as Java, to satisfy the demands of high-performance application areas. The basic idea is that the interpreter, from here on termed the virtual machine, monitors execution of a program at run-time. By profiling aspects of execution such as frequency of calls to a given method, the virtual machine can identify so called hotspots: regions of the program which are critical to performance. Once these regions are identified, the virtual machine invokes an aggressive re-compilation phase on them that results in either more efficient interpretable code or native code that can be executed directly by the host platform. By using the re-compiled code, execution can proceed more efficiently and without sacrificing portability of the original program. Java has benefited hugely from these sorts of techniques; earlier work on languages such as Self [10] and Smalltalk [12] has been harnessed to produce a number of high performance virtual machines such as the Sun Hotspot [24] and Jikes Research Virtual Machine [2] systems.

For all the positive aspects of this highly active research area, the basic premise that code written by the programmer might not be the same code executed by the host platform has some serious implications in terms of security. For example, one typically demands that the implementation of a cryptographic primitive exactly matches the specification so that security proofs can be valid.

Even when such proofs do hold, the security of cryptographic primitives is increasingly being attacked from a physical rather than mathematical point of view. For example, information leaked through so called side-channels can be collected by passive monitoring of execution features such as timing variation [17], power consumption [18] or electromagnetic emission [1]. Typically attacks consist of a collection phase which provides the attacker with profiles of execution, and an analysis phase which recovers the secret information from the profiles. Considering power consumption as the collection medium from here on, attack methods can be split into two main classes. Simple power analysis (SPA) is where the attacker is given only one profile and is required to recover the secret information by focusing mainly on the operation being executed. In contrast, differential power analysis (DPA) uses statistical methods to form a correlation between a number of profiles and the secret information by focusing mainly on the data items being processed. These passive attacks can be extended by considering active, fault injection methods whereby the attacker can physically manipulate the host platform to make it perform erroneous operations. A relevant example in terms of Java is the attack of Govindavajhala and Appel [15] who describe a method to subvert the Java type system by introducing transient errors into memory.

Both side-channel and fault injection attacks are commonly set in the context of devices such as smart-cards and embedded processors since they generally require that the attacker has physical access to the processing running the code under attack. The security risks here are exacerbated by the use of both types of device as conduits for sensitive financial or identity related information. However, Boneh and Brumley [8] and Bernstein [9] recently showed that timing attacks against remove servers are even possible without physical access to the target. The use of Java in both these areas hints at the problems one might expect. In particular, we motivate our work by noting the link between the fields of dynamic compilation and physical security that occurs as a result of the potential use of Just In Time (JIT) compilers on JavaCard based multi-application smart-cards. Examples include Jazelle enabled ARM processors which accelerate execution of Java in embedded contexts and include support for JIT compilation [3]. These types of JavaCard are widely used as a platform for rapid development of smart-card based applications; dynamic compilation allows the physical devices to remain simply and inexpensive while delivering acceptable levels of performance. As such, the security of these devices when used in combination with dynamic compilation is an important topic given the application areas they typically fill.

In this paper we investigate the specific question of whether dynamic compilation of Java programs can introduce side-channel vulnerabilities into the executed code where there were none in the original code. That is, in attempting to extract high levels of performance by transforming the program being executed, a dynamic compiler acting without knowledge of the domain could transform a secure program into an insecure one. We organise our work as follows. In Section 2 we present a concrete experiment on a some side-channel secured Java code that performs operations core to elliptic curve cryptography (ECC). Our

Algorithm 1: The *double-and-add* method for point multiplication.

Input : A point $P = (x_P, y_P)$ and an integer d
Output: A point $Q = (x_Q, y_Q) = d \cdot P$

$Q \leftarrow P$
for $i = |d| - 2$ **downto** 0 **do**
  $Q \leftarrow 2 \cdot Q$
  if $d_i = 1$ **then**
    $Q \leftarrow Q + P$
  end
end
return Q

Algorithm 2: The *double-and-add-always* method for point multiplication.

Input : A point $P = (x_P, y_P)$ and an integer d
Output: A point $Q = (x_Q, y_Q) = d \cdot P$

$Q[0] \leftarrow P$
for $i = |d| - 2$ **downto** 0 **do**
  $Q[0] \leftarrow 2 \cdot Q[0]$
  $Q[1] \leftarrow Q[0] + P$
  $Q[0] \leftarrow Q[d_i]$
end
return Q

experiments show that dynamic compilation acts to weaken the security of the implementation and as a result, may leak secret information to an attacker. As an attempt to resolve this highlighted problem, we investigate language and virtual machine based solutions in Sections 3 and 4. Finally, we offer some concluding remarks and highlight a number of interesting areas for future work in Section 5.

2 A Case Study: Elliptic Curve Cryptography

Restricting ourselves to working over the finite field $K = \mathbb{F}_{2^n}$ for some suitable n, an elliptic curve is defined by

$$E(K) : y^2 + xy = x^3 + Ax + B$$

for some parameters A and B. The set of rational points $P = (x, y)$ with $x, y \in K$ on this curve, together with the identity element $\mathcal{O}$, form an additive group under the so called chord-tangent group law. ECC based public key cryptography typically derives security by presenting an intractable discrete logarithm problem over this group. That is, one constructs a secret integer d and performs the operation $Q = d \cdot P$ for some public point P. Since reversing this operation is believed to be hard, one can then transmit Q without revealing the value of d.

Multiplication of a point by an integer is therefore a core operation in most ECC based systems; as detailed in Algorithm 1 it can be performed using the double-and-add method, an additive version of binary exponentiation. Note that we use d_i to denote the i-th bit of an integer d. Also note that for a random d, the point doubling operation will be used about twice as often as the point addition operation.

The actual point addition and doubling operations are often distinguishable from each other in a profile of execution because one is composed from a different sequence of finite field operations than the other. Using A and D to denote point addition and doubling respectively, the collection phase of an SPA attack presents the attacker with a profile detailing the operations performed during execution of the algorithm. For example, by monitoring execution of using the multiplier $d = 1001_2 = 9_{10}$, one obtains the profile

$$DDDA$$

Given this single profile, the analysis phase can recover the secret value of d simply by spotting where the point additions occur. If the sequence DA occurs during iteration i we have that $d_i = 1$ whereas if the sequence D occurs then $d_i = 0$.

One way to avoid this problem is to employ the double-and-add-always method due to Coron [11] and detailed in Algorithm 2. In this case, a dummy addition is executed if the real one is not. Although the cases where $d_i = 0$ and $d_i = 1$ are now indistinguishable, this method imposes a significantly performance penalty since many more additions are performed than is necessary. A more considered approach involves using the flexibility of the curve group law in terms of how the point addition and doubling operations can be implemented through different curve parameterisations, point representations and so on. For example, one can manipulate the formula for point addition and doubling so that they are no longer different. This is generally achieved by splitting the more expensive point addition into two parts, each of which is identical in terms of the operations it performs to a point doubling. Put more simply, instead of recovering the profile above from the SPA collection phase, an attacker gets:

$$XXXXX$$

where X represents an atomic, indestinguishable operation which could be a double or one step in an addition; from this the attacker gets no useful information other than about the Hamming weight and size of d.

Trichina and Bellezza [25] analyse the overhead and effectiveness of this approach using Jacobian projective coordinates on NIST standard curve over $K = \mathbb{F}_{2^{163}}$. Table 1 details their manipulated formula for point doubling and addition; note that C is a pre-computed constant equal to $\sqrt[4]{B}$. The doubling operation take a point $P = (x_P, y_P)$ and produces $R = (x_R, y_R) = 2 \cdot P$, the addition operation take points $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ and produces $R = (x_R, y_R) = P + Q$. Note that the each of the two addition steps has been made to use the same sequence of operations as the doubling formula by the inserting a total of eight dummy operations, these are denoted by use of register $\lambda_\perp$.

Doubling	Addition Step 1	Addition Step 2
$\lambda_1 \leftarrow z_P^2$	$\lambda_1 \leftarrow z_P^2$	$\lambda_{11} \leftarrow z_R^2$
$\lambda_3 \leftarrow y_P \cdot z_P$	$\lambda_2 \leftarrow \lambda_1 \cdot x_Q$	$\lambda_{12} \leftarrow z_R \cdot y_Q$
$z_R \leftarrow x_P \cdot \lambda_1$	$\lambda_3 \leftarrow \lambda_1 \cdot z_P$	$\lambda_{13} \leftarrow \lambda_8 \cdot \lambda_{10}$
$\lambda_4 \leftarrow x_P^2$	$\lambda_\perp \leftarrow \lambda_\perp^2$	$\lambda_{14} \leftarrow \lambda_7^2$
$\lambda_5 \leftarrow z_R + \lambda_4$	$\lambda_\perp \leftarrow \lambda_\perp + \lambda_\perp$	$\lambda_{15} \leftarrow \lambda_{11} + \lambda_{13}$
$\lambda_6 \leftarrow C \cdot \lambda_1$	$\lambda_6 \leftarrow y_Q \cdot \lambda_3$	$\lambda_{16} \leftarrow \lambda_7 \cdot \lambda_{14}$
$\lambda_7 \leftarrow x_P + \lambda_6$	$\lambda_7 \leftarrow x_P + \lambda_2$	$x_R \leftarrow \lambda_{15} + \lambda_{16}$
$\lambda_8 \leftarrow \lambda_4^2$	$\lambda_\perp \leftarrow \lambda_\perp^2$	$\lambda_\perp \leftarrow \lambda_\perp^2$
$\lambda_9 \leftarrow \lambda_8 \cdot z_R$	$z_R \leftarrow z_P \cdot \lambda_7$	$\lambda_{17} \leftarrow x_R \cdot \lambda_{10}$
$\lambda_{10} \leftarrow \lambda_5 + \lambda_3$	$\lambda_8 \leftarrow y_P + \lambda_6$	$\lambda_{18} \leftarrow \lambda_9 + \lambda_{12}$
$\lambda_{11} \leftarrow \lambda_7^2$	$\lambda_\perp \leftarrow \lambda_\perp^2$	$\lambda_\perp \leftarrow \lambda_\perp^2$
$x_R \leftarrow \lambda_{11}^2$	$\lambda_\perp \leftarrow \lambda_\perp^2$	$\lambda_\perp \leftarrow \lambda_\perp^2$
$\lambda_{12} \leftarrow \lambda_{10} \cdot x_R$	$\lambda_9 \leftarrow \lambda_8 \cdot x_Q$	$\lambda_{19} \leftarrow \lambda_{18} \cdot \lambda_{11}$
$y_R \leftarrow \lambda_9 + \lambda_{12}$	$\lambda_{10} \leftarrow z_R + \lambda_8$	$y_R \leftarrow \lambda_{17} + \lambda_{19}$

Table 1. Indistinguishable formula for elliptic curve point doubling and addition.

Method	With AOS	Without AOS
Doubling	0.03159 ms	0.80985 ms
Addition Step 1	0.02527 ms	0.81016 ms
Addition Step 2	0.03776 ms	0.80993 ms

Table 2. Timings for elliptic curve point doubling and addition with and without dynamic compilation.

Using these indistinguishable formula, we implemented an SPA resistant double-and-add based point multiplication program in Java and ran it on the Jikes Research Virtual Machine (RVM) [2]; our host platform incorporated a 2.8 GHz Pentium 4 processor. Using default options for the adaptive optimisation system (AOS) [4], RVM made an attempt to dynamically re-compile both the point doubling and addition methods as well as constituent methods for finite field arithmetic. As a product of aggressive inlining and subsequent optimisation, the timings in Table 2 show significant improvement, and slight skewing, of the unoptimised code.

The main point of note is that an execution profile of the optimised doubling method will differ radically from unoptimised version of the addition method. Even though we have gone to some lengths to make them indistinguishable from each other, the optimised versus unoptimised code will be distinguishable by virtue of their significantly different composition. Further, the time that re-compilation of these methods takes place is of interest. From the RVM log, we found that the doubling method was re-compiled at least 6 ms and as much as 20 ms before the addition methods depending on the value of d. Intuitively this makes sense: if the doubling method is called more often it is more likely to be quickly identified as a hotspot by either counter or sampling based monitors.

```
               double is re-compiled   addition is re-compiled
                        |                       |
 unoptimised double and | optimised double and  | optimised double and
 addition methods used  | unoptimised addition  | addition methods used
                        | methods used          |
                        v                       v
|-----------------------|-----------------------|-----------------------|
t0                      t1                      t2                      t3
                            ----- time ---->

 indestinguishable        destinguishable due     destinguishable due
                          to difference in        to skew introduced by
                          versions                optimisation
```

Fig. 1. A timeline of execution within three specific periods.

Given that the RVM carries out re-compilation in the background, one could expect a window whereby the optimised double method is in use at the same time as the unoptimised addition method.

Figure 1 details the general timeline of execution more simply. The point multiplication begins at time $t0$ with both the double and addition methods in an unoptimised form. Then, at time $t1$ the double method is identified as a hotspot and re-compiled; this leads to the double and addition methods now being distinguishable from each other due to their differing composition. At time $t2$ the addition methods are re-compiled but are still distinguishable from the double due to differing results from optimisation; this is highlighted by their differing timings after optimisation in Table 2. Execution concludes at time $t3$.

In short, the RVM adaptive optimisation system has introduced a side-channel vulnerability where there was none in the original code. Although the window of vulnerability may be small and may require some effort to mount a realistic attack against, information about bits of the secret value d can clearly be leaked during this period: one simply mounts a SPA type power analysis attack as described above and reads the bits of d directly from the operation trace during the vulnerable period. The seminal cryptanalytic work of Howgrave-Graham and Smart [16] shows that one can use this partial information, leaked for example from the execution of an ECDSA signing operation, to recover the whole secret. As such, one should view the execution of our test program as insecure.

3 Language Based Solutions

The natural and most simple way to address the highlighted problem is to empower the virtual machine with knowledge of side-channel properties relating to the program. That is, allow code fragments to be annotated with information that allows the virtual machine to avoid introducing vulnerabilities. There has already been extensive research into annotating Java class files with information

```java
class point
{
  @nojit
  public void dbl( point p )
  {
    ...
  }
}

class point
{
  nojit public void dbl( point p )
  {
    ...
  }
}

class point
{
  public void dbl( point p )
  {
    nojit
    {
      ...
    }
  }
}
```

(a) Basic annotation via mark-up and keywords to prevent dynamic compilation.

```java
class point
{
  @ensurejitmatches( "add_step2" )
  public void add_step1( point p,
                         point q )
  {
    ...
  }

  @ensurejitmatches( "add_step1" )
  public void add_step2( point p,
                         point q )
  {
    ...
  }
}
```

(b) Semantically richer mark-up based annotation to guide dynamic compilation.

Fig. 2. A sketch of two methods for annotating Java source code to inform the JVM of side-channel related properties.

to improve performance; for example [19, 5]. This sort of work typically packages information into attributes within the class data structure as prescribed by the Java language specification [14]. By providing hints about statically collectible information, for example control flow or register allocation, the workload of dynamic compilation can be significantly reduced. Clearly it is trivial to implement a similar system to allow annotation of classes with side-channel related attributes that can be passed to the virtual machine. At the language level, information relating to side-channel security is easily accommodated by the existing Java annotation mechanism [14][Chapter 9.7]. Figure 2 demonstrates two methods by which this information could be harvested by the compiler for injection into the class file.

Figure 2a uses basic annotation that instructs the dynamic compiler to leave the associated method alone. This could be a severe trade-off in the sense that by prohibiting dynamic compilation of these methods, performance is sure to suffer. However, there are areas other than security in which it may also be desirable to turn off dynamic compilation for a particular code fragment; predictable and real-time computing for example. One could achieve this via either a mark-up based annotation or by using a new or existing keyword. The former approach has the advantage of not interfering with existing language and compiler definitions; the latter has the advantage of being able to associate with more fine grained regions of code, much like the `synchronized` keyword, rather than just the class declarations.

Figure 2b uses a richer, more relaxed approach, specifying that dynamic compilation is permitted as long as the compiler can satisfy that the named methods still match each other. This offers some hope of a compromise between performance and security yet opens an interesting question as to how such a joint-compilation phase might proceed. The task of detecting any mismatch between the results of re-compilation, and subsequent removal of said mismatches [7], has been formalised by Molnar et al. [21]. Their program counter model provides an ideal framework for the dynamic compiler to verify the act of optimisation satisfies the security constraints passed via annotation by the programmer.

4 Virtual Machine Based Solutions

Micali and Reyzin present a theoretical model for reasoning about side-channel attacks [20]; essentially this requires a processing device with well defined, if slightly impractical, properties. Even so, it offers a context in which one can prove a primitive secure against attack and as such, one could imagine trying to augment the virtual machine to match their requirements. The major sticking point in doing this is axiom four in which Micali and Reyzin state

> *The information that may be leaked by a physical observable device is the same in any execution with the same input, independent of the computation that takes place before the device is invoked or after it halts.*

This seems to preclude any form of adaptive approach for dynamic compilation outright; the axioms of Micali and Reyzin do not seem to cover the case where code is altered at run-time so the case for dynamic compilation is equally unclear. In short, even if the proposed device were realistic the resulting security proofs would be invalid when considered in the context of a virtual machine with dynamic compilation.

However, as a general virtual machine based solution, one could clearly side-step the problem by demanding that the virtual machine re-compile all code before execution begins, perhaps using performance oriented annotation hints mentioned previously [19, 5]. However, this seems a perverse approach since it removes any benefit that could be achieved by a dynamic system over a static alternative.

5 Conclusions

The premise that under systems using dynamic compilation code executed might not match the code written should be of concern to practitioners of security conscious, side-channel aware implementation of cryptography. Although formal models of physical security are difficult to construct, most standard defence methods assume a conventional, native model of execution to ensure their success. We have shown that dynamic compilation, and dynamic execution in general, breaks this assumption and can therefore present vulnerabilities even when said defences are implemented.

To some extent, resolution of the highlighted problem is related to the concept of certified compilation [23] in the sense that to solve it we demand the compiler satisfy some formal requirements of the transformations it uses. Our results are, in a sense, a simple re-phrasing of an existing problem in this area. That is, it has long been known that static compilation might yield code after optimisation which doesn't give the same security properties as one might expect. Indeed, after some thought it is hard to come up with realistic examples which are specific to dynamic compilation. With this in mind, it seems vital for both static and dynamic compilers to be aware of security and cryptography in *all* aspects of their operation [6].

Our work represents a preliminary investigation into this area; at least two strands of further work seem interesting and important.

Proof Carrying Code Approaches It seems interesting to investigate how one might embed a proof of the side-channel properties of a program within itself; this essentially extends the idea of a proof carrying code (PCC) [22] to the context of side-channel security. The problem of constructing such a proof and a compiler capable of certifying it has met the requirements seems difficult if only because formal models of physical security are not as mature as in other areas [13, 20].

Other Dynamic Execution Features The cryptographic community has already explored the problems associated with data-dependant behaviour of the memory and cache subsystem; see for example [26, 9]. Dynamic execution and memory management as used in Java could offer similar data-dependant behaviour. For example, it seems possible that code which seems secure leaks information which executed on this sort of platform; management of heap objects and their garbage collection offers a similarly observable feature as cache behaviour for example. With this in mind, it seems interesting to examine other aspects of interpreted execution whose operational semantics might differ from those expected by programmers implementing standard side-channel defences.

6 Acknowledgements

The author would like to thank Nigel Smart and Andrew Moss for their input throughout the duration of this work.

References

1. D. Agrawal, B. Archambeault, J.R. Rao and P. Rohatgi. The EM Side-Channel(s). In *Cryptographic Hardware and Embedded Systems (CHES)*, Springer-Verlag LNCS 2523, 29–45, 2002.
2. B. Alpern, C.R. Attanasio, J.J. Barton, M.G. Burke, P Cheng, J-D. Choi, A. Cocchi, S.J. Fink, D. Grove, M. Hind, S.F. Hummel, D. Lieber, V. Litvinov, M.F. Mergen, T. Ngo, J.R. Russell, V. Sarkar, M.J. Serrano, J.C. Shepherd, S.E. Smith, V.C. Sreedhar, H. Srinivasan and J. Whaley. The Jalapeño Virtual Machine. In *IBM System Journal*, **39**(1), 2000.
3. ARM. Jazelle White Paper. Available at `http://www.arm.com/pdfs/JazelleWhitePaper.pdf`.
4. M. Arnold, S.J. Fink, D. Grove, M. Hind and P.F. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2000.
5. A. Azevedo, A. Nicolau and J. Hummel Java Annotation-aware Just-In-Time (AJIT) Compilation System. In *Java Grande Conference*, 142–151, 1999.
6. M. Barbosa, R. Noad, D. Page and N.P. Smart. First Steps Toward a Cryptography-Aware Language and Compiler. In *Cryptology ePrint Archive*, Report 2005/160, 2005.
7. M. Barbosa and D. Page. On the Automatic Construction of Indistinguishable Operations. In *Cryptography And Coding*, Springer-Verlag LNCS 3796, 233–247, 2005.
8. D. Boneh and D. Brumley. Remote Timing Attacks Are Practical. Available at `http://crypto.stanford.edu/~dabo/papers/ssl-timing.pdf`.
9. D.J. Bernstein. Cache-timing Attacks on AES. Available at `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.
10. C. Chambers and D. Ungar. Making Pure Object-orietned Languages Practical. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1–15, 1991.
11. J-S. Coron. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In *Cryptographic Hardware and Embedded Systems (CHES)*, Springer-Verlag LNCS 1717, 292–302, 1999.
12. L.P. Deutsch and A.M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Principles of Programming Languages (POPL)*, 297–302, 1984.
13. R. Gennaro, A. Lysyanskaya, T. Malkin, S. Micali and T. Rabin. Algorithmic Tamper-Proof (ATP) Security: Theoretical Foundations for Security against Hardware Tampering. In *Theory of Cryptography*, Springer-Verlag LNCS 2951, 258–277, 2004.
14. J. Gosling, B. Joy, G. Steele and G. Bracha. The Java Language Specification, Third Edition. Addison-Wesley, 2005.
15. S. Govindavajhala and A.W. Appel. Using Memory Errors to Attack a Virtual Machine. *IEEE Symposium on Security and Privacy*, 154–165, 2003.

16. N. Howgrave-Graham and N.P. Smart. Lattice Attacks on Digital Signature Schemes. *Designs, Codes and Cryptography*, **23**, 283–290, 2001.
17. P.C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology (CRYPTO)*, Springer-Verlag LNCS 1109, 104–113, 1996.
18. P.C. Kocher, J. Jaffe and B. Jun. Differential Power Analysis. In *Advances in Cryptology (CRYPTO)*, Springer-Verlag LNCS 1666, 388–397, 1999.
19. C. Krintz and B. Calder Using Annotation to Reduce Dynamic Optimization Time. In *Programming Language Design and Implementation (PLDI)*, 156–167, 2000.
20. S. Micali and L. Reyzin. Physically Observable Cryptography (Extended Abstract). In *Theory of Cryptography*, Springer-Verlag LNCS 2951, 278–296, 2004.
21. D. Molnar, M. Piotrowski, D. Schultz and D. Wagner. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *Cryptology ePrint Archive*, Report 2005/368, 2005.
22. G.C. Necula. Proof-Carrying Code. In *Principles of Programming Languages (POPL)*, 106–119, 1997.
23. G.C. Necula and P. Lee. The Design and Implementation of a Certifying Compiler. In *Programming Language Design and Implementation (PLDI)*, 333–344, 1998.
24. Sun Microsystems. Java Hotspot Whitepaper. Available at `http://java.sun.com/products/hotspot/`.
25. E. Trichina and A. Bellezza. Implementation of Elliptic Curve Cryptography with Built-In Counter Measures against Side Channel Attacks. In *Cryptographic Hardware and Embedded Systems (CHES)*, Springer-Verlag LNCS 2523, 98–113, 2002.
26. Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri and H. Miyauchi. Cryptanalysis of DES Implemented on Computers with Cache. In *Cryptographic Hardware and Embedded Systems (CHES)*, Springer-Verlag LNCS 2779, 62–76, 2003.