Random Register Renaming to Foil DPA

D. May, H.L. Muller, and N.P. Smart

Department of Computer Science,
Woodland Road, University of Bristol, BS8 1UB, UK
{dave,henkm,nigel}@cs.bris.ac.uk

Abstract. Techniques such as DPA and SPA can be used to find the secret keys stored in smart-cards. These techniques have caused concern for they can allow people to recharge their stored value smartcards (in effect printing money), or illegally use phone or digital TV services. We propose an addition to current processors which will counteract these techniques. By randomising register usage, we can hide the secret key stored in a smartcard. The extension we propose can be added to existing processors, and is transparent to the algorithm.

1 Background

Modern cryptography is about ensuring the integrity, confidentiality and authenticity of digital communications. As such it has a large number of applications from e-commerce on the Internet through to charging mechanisms for pay–per-view-TV. As more and more devices become network aware they also become potential weak links in the chain. Hence cryptographic techniques are now being embedded into devices such as smart cards, mobile phones and PDA's. This poses a number of problems since the cryptographic modules are no longer maintained in secure vaults inside large corporations. For a cryptographic system to remain secure it is imperative that the secret keys used to perform the required security services are not revealed in any way.

The fact that secret keys are now embedded into a number of devices means that the hardware becomes an attractive target for hackers. For example if one could determine the keys which encrypt the digital television transmissions, then one could create decoders and sell them on the black market. On a more serious front if one could determine the keys which protect a number of stored value smart cards, which hold an electronic representation of cash, then one could essentially print money.

Since cryptographic algorithms themselves have been studied for a long time by a large number of experts, hackers are more likely to try to attack the hardware and system within which the cryptographic unit is housed. A particularly worrying attack has been developed in the last few years by P. Kocher and colleagues at *Cryptography Research Inc.*, see [6] and [7]. In these attacks a number of physical measurements of the cryptographic unit are made which include power consumption, computing time or EMF radiations. These measurements are made over a large number of encryption or signature operations and then,

Ç.K. Koç, D. Naccache, and C. Paar (Eds.): CHES 2001, LNCS 2162, pp. 28–38, 2001.
© Springer-Verlag Berlin Heidelberg 2001

using statistical techniques, the secret key embedded inside the cryptographic unit is uncovered.

These attacks work because there is a correlation between the physical measurements taken at different points during the computation and the internal state of the processing device, which is itself related to the secret key. For example, when data is loaded from memory, the memory bus will have to carry the value of the data, which will take a certain amount of power depending on the data value. Since the load instruction always happens at the same time one can produce correlations between various runs of the application, eventually giving away the secret of the smart card. The three main techniques developed by Kocher et. al. are timing attacks, simple power analysis (SPA) and differential power analysis (DPA). It is DPA which provides the most powerful method of attack, which can be mounted using very cheap resources.

Following Kocher's papers a number of people have started to examine this problem and propose solutions, see [1], [2], [3] and [8]. Goubin and Patarin [3] give three possible general strategies to combat DPA type attacks:

1. Introduce random timing shifts so as to decorrelate the output traces on individual runs.
2. Replace critical assembler instructions with ones whose signature is hard to analyse, or reengineer the crucial circuitry which performs the arithmetic operations or memory transfers.
3. Make algorithmic changes to the cryptographic primitives under consideration.

In [9] May, Muller and Smart propose a method for introducing highly aggressive randomised execution into a conventional processor. They argue that this produces a great deal of temporal misalignment of traces, which can help defeat DPA. The methodology is to take standard techniques from the design of super-scalar architectures and replace parallel execution with random execution. They call this new processor architecture NDISC for *Non-Deterministic Instruction Stream Computer*.

This defence essentially combines all three of the above defences in that it adds considerable timing shifts to the instructions, it introduces circuitry which is hard to analyse and essentially makes algorithmic changes to the program "on the fly".

In this paper we expand on this philosophy by proposing a technique which allows the non-deterministic altering of the register to register or memory to register transfers. As such this produces a defence more akin to the second of the proposed defences above. This new defence can be implemented using a very small number of changes to the underlying processor and is completely transparent to the algorithm.

In super-scalar architectures, see [4], [5] or [10], it is standard practice to implement a form of register renaming. This allows the processor to schedule more instructions in parallel. If such a system was implemented in the NDISC processor then the instruction stream could be executed in an even greater randomised order. However, if the register renaming was performed in a randomised, rather

than the standard deterministic manner, then one would obtain the extra effect of the power consumed by each register write operation would be different from one run to the next.

The concept of randomised register renaming can be implemented in a conventional processor to obtain some defence against DPA, but when implemented in an NDISC processor we expect the overall defence against DPA to be greatly enhanced. Hence, we first present an introduction to what an NDISC processor is.

2 NDISC Processors

In order to prevent attacks based on correlating data, we have designed a simple addition to standard processors that randomises instruction issuing [9]. Crucially, an attack works because two runs of the same program give comparable results; everything compares bar the data. By changing the data even slightly the attacker will get a knowingly different trace, and by correlating the traces, one builds a picture of what is happening inside the processor.

An NDISC processor removes correlation between runs, thereby making the attack much harder. A conventional processor executes a sequence of instructions deterministically; it may execute instructions *out-of-order*, but it will always execute instructions out-of-order in the same way. If the same program is run twice in a smart card, then the same instruction trace will be executed. By allowing the processor to at run time choose a random instruction ordering, we get multiple possible traces that are executed.

2.1 Random Issuing

In single pipeline processors a sequence of instructions is executed in the order in which they are fetched by the processor. There is a little out-of-order execution to help with branch prediction but this all occurs on a very small scale. On multiple pipeline processors there are a number of *execution units* through which independent instructions can be passed in parallel. For example, if a processor has a logic pipeline and an integer-arithmetic pipeline, then the following two instructions

```
ADD R0, R1, R1
XOR R4, R5, R5
```

may be executed in parallel in the two pipelines. One pipeline will execute the ADD, the other will execute the XOR.

Our idea is the following: like a superscalar we identify instructions that can be issued independently, but instead of using this information to issue instructions in parallel, we use this information to execute instructions out-of-order, where the processor makes a random choice as to issue order. We call this process *Instruction Descheduling*. This creates a level of non-determinism in the internal workings of the processor. This is illustrated in Figure 1.

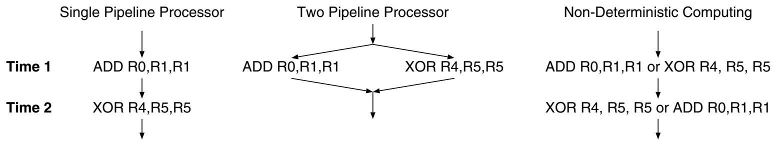

Fig. 1. Simple comparison of how a Non-deterministic processor executes two instructions as opposed to other processors

The reduction in the effectiveness of DPA results from the fact that the power trace from one run will be almost completely uncorrelated with the power trace from a second run, since on the two runs different execution sequences are used to produce the same result.

Instruction descheduling means that at run time the processor will select, at random, an instruction to execute, thereby randomising the instruction stream, and randomising the access pattern to memory caused by both data and instruction streams.

A full description of how an NDISC machine can be implemented is discussed in [9], we outline the NDISC architecture here. The set of instructions waiting to be executed is held in a block called an issue window. The random instruction selection unit randomly selects instructions from the issue window that are executable. An instruction is considered executable if it does not depend on any result that is not yet available, and the instruction does not overwrite any data that is still to be used by other instructions that are not yet executed, or instructions that are in execution in the pipeline.

The implementation of this closely follows the implementation of multi-issue processors. There is a block of logic that determines conflicts between instructions, resulting in a set of instructions that is executable. From this set we select an instruction at random. Given a random number generator, which will normally be constructed from a pseudo random number generator that is reseeded regularly with some entropy, we select one of the executable instructions and schedule it for execution.

3 Random Register Renaming

3.1 Basic Register Renaming

Register renaming is a common technique used to improve the performance of processors. Renaming works by defining a set of virtual register identifiers (which are used in the instruction set of the processors) and a set of physical registers (which are used in the execution unit). At any moment in time, each virtual register identifier is associated uniquely with a physical register identifier.

This binding is unique at any time, but the strength of renaming is that it changes with time. Any time that a virtual register is overwritten, the binding between the virtual and physical register can be severed, and a fresh physical register can be assigned to the virtual register. The reason that this increases

performance in a standard processor is that this physical register can be used immediately for storing new values, whereas the old physical register can still hold values that are used by instructions which are still in execution. This allows out of order parallel execution.

As an example, consider the following bit of code which implements the two assignments `A=B ; C=D ;`:

```
LOAD     B, R0
STORE    R0, A
LOAD     D, R0      ; R0 is overwritten
STORE    R0, C
```

Upon execution of the code the third instruction will overwrite the value of register R0 with the value of D. This instruction can only be executed if the previous store instruction has been completed. If the registers are renamed so that we use R1 instead of R0, we would get the following code:

```
LOAD     B, R0
STORE    R0, A
LOAD     D, R1      ; R1 is overwritten
STORE    R1, C
```

In this code segment the third line of code can be executed in parallel with the first two, speeding up the processor. Static register allocation (by the compiler) does not achieve the same effect as register renaming for two reasons. First with register renaming one needs less bits in the instruction set to encode registers. Second, register renaming works at run time; register assignments are based on which instructions are in progress, and which are waiting for execution.

3.2 Random Register Renaming

We employ *Random Register Renaming* in order to weaken a DPA attack. Our observation is that a large fraction of power trace is produced by overwriting register values. Each time a value of a register is overwritten, the power consumption is related to the number of bits that are flipped. We are going to rename registers non deterministically; that way, any time that a register is overwritten, it overwrites a non predetermined value, randomising the power trace.

As an example, consider the following code:

```
LOAD  B,R0
STORE R0,A
LOAD  D,R0
STORE R0,C
```

Where we suppose that we have just one virtual register identifier (R0) and we have two physical registers (Reg0, Reg1). When this piece of code is executed there will be four different ways to execute it:

```
LOAD  B,Reg0     LOAD  B,Reg0     LOAD  B,Reg1     LOAD  B,Reg1
```

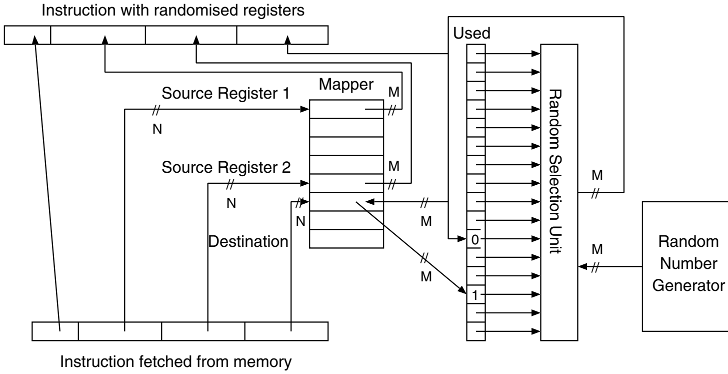

Fig. 2. Random register renaming for simple processor

```
STORE Reg0,A      STORE Reg0,A      STORE Reg1,A      STORE Reg1,A
LOAD  D,Reg0      LOAD  D,Reg1      LOAD  D,Reg0      LOAD  D,Reg1
STORE Reg0,C      STORE Reg1,C      STORE Reg0,C      STORE Reg1,C
```

Each of these execution traces has its own individual power-trace. Indeed, for longer instruction sequences the number of possible traces grows exponentially, and combined with NDISC execution [9], we can attain high levels of protection.

4 Implementation

4.1 Basic Implementation

In order to implement random register renaming we distinguish between virtual register identifiers (these are the registers as specified by the instruction set), and physical registers (which are the ones used by the processor). We assume that the number of physical registers is greater or equal to the number of virtual register identifiers. In particular we assume that there are 2^V virtual register identifiers and 2^P physical registers, where P is larger than V.

An instruction is fetched into the processor and preprocessed to rename its registers. The registers are renamed using a Virtual to Physical mapping table. This process is illustrated in Figure 2. The register mapper maintains a mapping from virtual identifiers to physical registers, this can be seen as an array of register values. A series of bits called `used` maintains whether a physical register is at present in use.

Each instruction has a number source and destination operands. In most instructions these operands are virtual register identifiers. On an instruction prefetch, the virtual register identifiers of the source operands are mapped onto physical registers using the virtual-to-physical mapper. This is done by using the virtual register number as an index in the mapping table, and thus locating a physical register number.

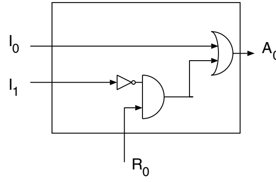

Fig. 3. Selecting a random register from a set of 2 (R-box^0)

The physical register for the destination operand is selected from the set of available free physical registers using a run time calculated random value as shown in the next section. The virtual register identifier of the destination operand was mapped onto another physical register. This physical register is now marked as free. The physical register that has been selected as the destination operand is marked as used and the the mapping table is updated associating the randomly selected physical register with the virtual identifier of the destination operand. Note that in a pipelined processor the register will only be marked free later; this is common practice in register renaming hardware. When source and destination registers have been mapped, the remapped instruction can be passed to the issue mechanism.

4.2 Selecting a Random Register from a Set

It is essential that in a single clock cycle we extract a free register. Generating a random number in a clock cycle is not difficult, but we must only pick from the set of *free* registers. For this purpose, we have devised a random selection unit. The random selection unit is a tree structure. Its simplest form consists of a single "R-box^0" selecting one random element from a set of 2, shown in Figure 3.

This box has two inputs I_0 and I_1 which denote whether register 0 and register 1 are currently in use, a random input bit R_0, and an output-bit A_0 pointing to the randomly selected free register. If I_0 and I_1 are both zero (i.e., both registers are free), then the random bit determines whether the output is zero or one. If one of the registers is not free, then the output is fixed to point to the other register.

We now construct a tree of AND-gates, AND-ing the used-bits of each register as shown in Figure 4. The output of each of the AND gates is '0' if there is at least one free register in the set of registers covered by that AND gate, and '1' if there are no free registers in that set. So, the output of the left-hand top-level AND gate is '0' if there is at least one free register in the lower half of the registers. The output of the right-hand top-level AND gate is '0' if there is at least one free register in the higher half of the registers. We use R-box^0 to determine which of those two halfs of the register set we are going to pick a register from.

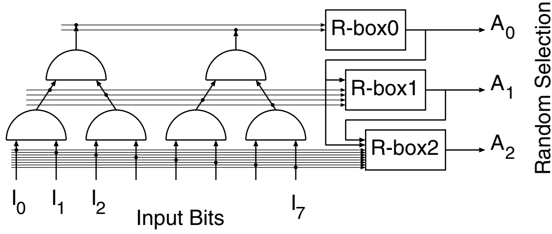

Fig. 4. Tree to reduce free set

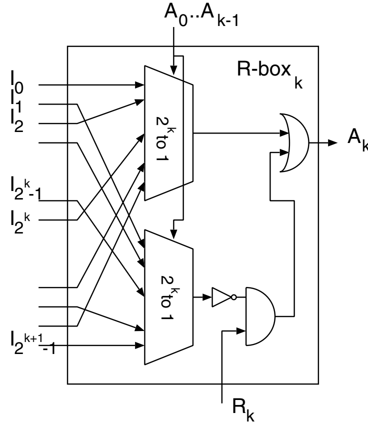

Fig. 5. General R-box

At the next level down in the tree we have outputs of four AND gates, which outputs state whether there are free registers in each of the four quarters of the register bank. R-box^0 has determined whether the free register is picked from the top or the bottom half, and R-box^1 is going to decide which of the two quarters to use.

We generalise R-box^1, and R-box^k from R-box^0 to have 2^{k+1} inputs $I_0, \ldots,$ $I_{2^{k+1}}$, k input-address bits $A_0, \ldots, A_{k-1}$ and one output bit A_k, as shown in Figure 5. This R-box selects a value for A_k so that bit $I_{A_0,\ldots,A_k}$ is zero. The selection is based on the assumption that all the higher address bits ($A_0 \ldots A_k$, the inputs provided by previous R-boxes) have been selected before in such a way that there is an empty register available. The selection process works by selecting two bits from the set $I_0, \ldots, I_{2^{k+1}}$ using $A_0 \ldots A_{k-1}$, and subsequently uses the random bit to decide which of those two bits to select. This gives us one more address bit, which results in the picking of a single input $I_{A_0,\ldots,A_k}$.

This architecture works since we can guarantee to have at least one physical register free at any point in the program. This solution selects a random bit which is set to zero in a constant time; at the cost of the random selection being skewed. In particular, if all but one of the bits in one half of the set are one,

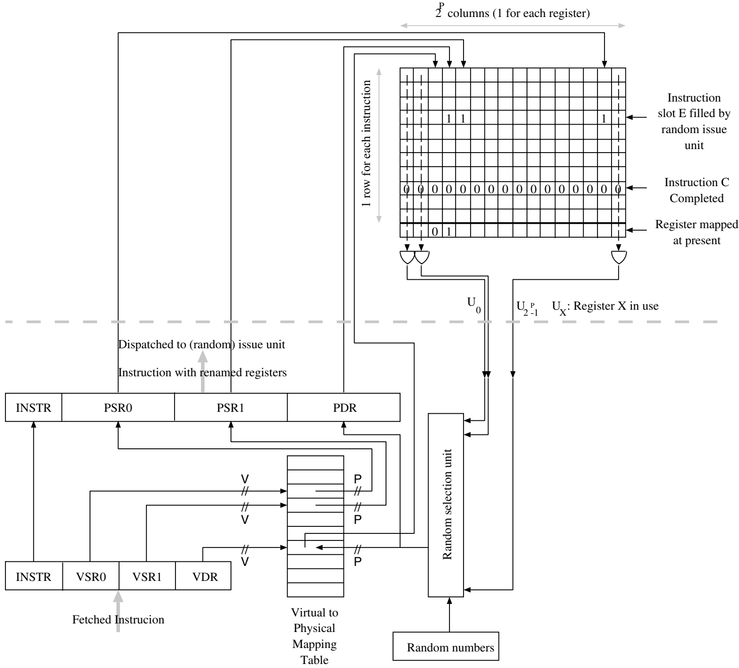

Fig. 6. Register renaming in conjunction with (non deterministic) out of order execution

than this solution will favour that solution with a 50% probability. This can be improved by performing a random rotation on the input bits prior to selecting a random bit.

4.3 Random Register Renaming in a Non Deterministic Processor

Random register renaming can be applied to any processor. For a standard microprocessor we have shown that random register renaming requires very little alteration to the standard renaming unit found in modern super-scalars. When attaching this unit to a NDISC processor [9], one needs to be more careful. For NDISC processors we determine the set of free registers as shown in Figure 6. A matrix of size $I \times 2^P$ (where I is the issue window size) maintains for each instruction in the issue window which registers that instruction uses. A one in element (i, p) in the matrix indicates that register p will be used by instruction i. A zero in element (i, p) indicates that register p will not be used by instruction i. When an instruction is stored in location E of the issue buffer, the bits in row E associated with its source and destination registers are set to one. An extra row at the bottom of the matrix keeps track of which registers are currently mapped and could, therefore, be used by future instructions.

When an instruction C has been dispatched and completed, the bits for row C are reset to zero (an optimisation would be to reset the source registers when the source registers have been read by the execution unit, and to reset the destination register when the results value has been written). The logical OR of all bits in a column r determines whether a register r is in use, a one value indicating that the register is used. These values are used by the renaming unit in order to determine which register to use in order to remap a destination register, as in the previous example of a standard processor.

5 Conclusion and Future Work

We have shown how one can use ideas from super-scalar architectures to produce a randomised *non-deterministic* processor. This idea essentially allows a single instruction stream to correspond to one of an exponential number of possibly executed programs, all of which produce the same output.

In the case of random register renaming used on its own the actual executed program is in the same program sequence but with the source and destination registers randomly altered. In the case of using both random register renaming and the ideas in [9] we obtain not only a program whose registers have been randomly reassigned, but the program sequence is now randomised as well. Since attacks such as differential power analysis rely on the correlation of data across many execution runs of the same program, the idea of random register renaming can help to defeat such attacks.

Further research still needs to be carried out, yet we feel that the proposed solution to differential power analysis gives a number of advantages; such as the fact that program code does not need to be modified and neither speed nor power consumption are compromised. In the near future we aim to produce a demonstration version of an the NDISC processor with the random register renaming included in the main execution unit. This demonstration will then be tested for resistance to DPA on a number of cryptographic algorithms.

References

1. S. Chari, C.S. Jutla, J.R. Rao and P. Rohatgi. Towards sound approaches to counteract power-analysis attacks. *Advances in Cryptology, CRYPTO '99*, Springer LNCS 1666, 398–412, 1999.
2. S. Chari, C.S. Jutla, J.R. Rao and P. Rohatgi. A cautionary note regarding evaluation of AES candidates on Smart-Cards. *Second Advanced Encryption Standard Candidate Conference*, Rome March 1999.
3. L. Goubin and J. Patarin. DES and differential power analysis. The "duplication method". *Cryptographic Hardware and Embedded Systems*, Springer LNCS 1717, 158–172, 1999.
4. J.L. Hennessy and D.A. Patterson. *Computer architecture: a quantitative approach.* Morgan Kaufmann Publishers, Palo Alto, California, 1990.
5. N. P. Jouppi and D. W. Wall. *Available instruction-level parallelism for superscalar and super-pipelined machines.* ASPLOS-III, 272–282, 1989.

6. P. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS and other systems. *Advances in Cryptology, CRYPTO '96*, Springer LNCS 1109, 104–113, 1996.
7. P. Kocher, J. Jaffe and B. Jun. Differential Power Analysis. *Advances in Cryptology, CRYPTO '99*, Springer LNCS 1666, 388–397, 1999.
8. O. Kömmerling and M. Kuhn. Design Principles for Tamper-Resistant Smartcard Processors. *USENIX Workshop on Smartcard Technology*, Chicago, Illinois, USA, May 10-11, 1999.
9. D. May, H. Muller and N.P. Smart Non-Deterministic Processors To appear *ACISP 2001*, Springer Verlag, LNCS, July 2001.
10. D Sima, T Foutain and P Kacsuk. *Advanced Computer Architectures.* Addison Wesley, 1997.