

Instruction Stream Mutation for Non-Deterministic Processors

J. Irwin D. Page N.P. Smart

Department of Computer Science, University of Bristol, UK

{jimbob, page, nigel}@cs.bris.ac.uk

Abstract

Differential power analysis (DPA) has become a real-world threat to the security of cryptographic hardware devices such as smart-cards. By using cheap and readily available equipment, attacks can easily compromise algorithms running on these devices in a non-invasive manner. Adding non-determinism to the execution of cryptographic algorithms has been proposed as a defence against these attacks. One way of achieving this non-determinism is to introduce random additional operations to the algorithm which produce noise in the power profile of the device. We describe the addition of a specialised processor pipeline stage which increases the level of potential non-determinism and hence guards against the revelation of secret information.

1. Introduction

Several ideas which act to reduce the effectiveness of differential power analysis [10] (DPA) attacks against cryptographic hardware devices have been proposed [2, 4]. Any defence against DPA is valuable since it allows attackers to liberate secret information from the device in a non-invasive and cost effective manner. DPA attacks are mounted by measuring the power usage of a cryptographic device, such as a smart-card, and correlating this information with features in the algorithm source code. By using suitable statistical techniques on a large set of power profiles, the attacker can read secret information, such as the users private key, as easily as if it were public. Far from being a theoretical attack, DPA has been successfully carried out against existing devices [1, 15] and is therefore a major threat to secure systems.

Perhaps the most effective defence against DPA attacks is the introduction of a new processor architecture which uses instruction level parallelism in the algorithm to execute the resulting instruction stream in a non-deterministic, random order [13]. By removing the correlation between features in the DPA profile and the algorithm source code, a non-deterministic processor makes retrieving useful information significantly harder. Non-deterministic execution can also be introduced by adding random additional calculations to the algorithm which add noise to the resulting DPA trace [15]. This acts to hide useful information from the attacker and, if done at random points during the run-time of the algorithm, can be difficult to spot and remove. The problem with adding such operations is that they must be careful not to alter the meaning of the real instructions being executed and hence not alter the outcome of the algorithm. Adding enough operations while retaining this property is made harder since they must be inserted at run-time so that the attacker is unable to spot them, even if they are in some way obfuscated [6], in the public source code.

This paper describes how a non-deterministic processor may be augmented by an additional pipeline stage, called a *mutation unit*, which implements ideas based around adding random calculations to the executed algorithm. The purpose of this specialised hardware device is to aggressively

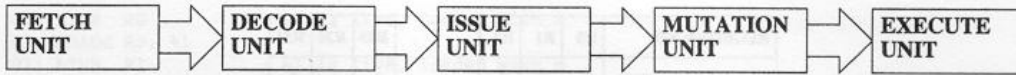


Figure 1. Pipeline stages of a non-deterministic processor.

alter the instruction stream such that the meaning of a program is retained while the instructions used to implement it change radically each time it is run. This is achieved by using knowledge of register liveness, generated by the compiler, to allow the processor to alter register usage and mapping patterns at run-time. The overall solution means that the composition, as well as the order, of the program will change on a per-run basis making DPA even harder to perform.

The implementation of instruction stream mutation draws from work conducted to construct liveness analysis from executable images [16] and carrying liveness information as well as other high level structures through the compiler tool-chain [12, 17]. Looking at this work, it is clear that using liveness analysis in novel ways to perform aggressive, cross module optimisation at compile, link and run-time has already been explored to some extent. In addition, the outcome of the instruction stream mutator is similar to that produced by software tamper-proofing [11, 18] systems which act, at compile-time, to obfuscate a source program. By harnessing both these ideas and allowing the liveness information to guide the operation of the running processor, we can approach the problem of reducing the effectiveness of DPA attacks in a way that would be otherwise impossible.

Sections 2 and 3 of the paper describe the additions to, and alterations of, hardware and software elements of an existing processing device. The main resulting difference is the addition of hardware resident liveness information which keeps track of register liveness state. Section 4 describes how these additional hardware and software elements may be combined to create a mutation unit capable of changing the instruction stream and increasing the level of misalignment between program source code and resulting DPA profile.

2. Hardware

The mutation unit is located directly before the execution unit in the pipeline of a non-deterministic processor. By adopting this position, as shown in Figure 1, the unit can be sure that any previous decoding and random issue of incoming instructions will have already been performed. The unit may therefore examine and operate on each instruction before dispatching it to the execution unit.

The ability for the mutation unit to alter the register usage and mapping patterns of incoming instructions depends on the presence of a table that describes when registers contain useful values and when they may be safely over-written. This table, which is resident in the mutation unit and named the *liveness table*, describes the liveness status, as determined by the compiler or programmer, of values in physical registers. Given that a single bit can be used to determine if a given register contains either a live or dead value, the liveness table can take the form of a bitfield. In this scheme, bit n in the bitfield denotes that register number n is live if set to 1 and dead if set to 0 as demonstrated by Figure 2.

Since the liveness table is resident in the mutation unit and not as a general purpose addressable register, it must be kept valid by the use of two extra instruction set architecture (ISA) level operations. These operations, the *LIVE* and *DEAD* instructions, act to set or clear bits in the liveness table that correspond to the registers named as their arguments. For example, the instruction *LIVE R0* sets the first bit of the liveness table, indicating that register *R0* is live. The mutation unit can spot

REGISTER NO.	R0	R1	R2	R29	R30	R31
LIVENESS STATUS	1	0	0	1	1	0

Figure 2. A liveness table.

```

00: int a, b, c, d, e;
01:
02: a = 1;
03: b = 2;
04: c = a + b;
05: d = c * b;
06: e = 3;

```

Figure 3. A simple example source code fragment.

these instructions as they pass through the pipeline and use the information to update the liveness table.

It is important to note that although this scheme is typical, there are a number of alternatives which may be appropriate in certain circumstances. Firstly, in the case where the execution and mutation units are effectively a single entity, the liveness table can be represented as a special purpose register and updated using general purpose mask and shift instructions. For example, if the register is named *LT* then the instruction *XOR LT, LT, %1* might be used to toggle the first bit of the table and convey the information that the liveness of *R0* has changed. Secondly, while this document describes the *LIVE* and *DEAD* instructions as taking single register arguments, it is possible to set or clear more than one bit in the liveness table at once by using register masks. In this scheme each instruction takes a list of registers which is used to generate the mask used as the instruction argument. For example, *LIVE R0, R1, R3* might generate the mask *0x0B* which is encoded as *LIVE 0x0B* to set registers *R0, R1* and *R2* live in one operation. This optimisation may be performed by either the compiler or, more likely the assembler, using conventional peephole-optimisation techniques.

3. Software

In order to ensure that correct program behaviour is retained when the mutation unit alters the instruction stream, the liveness table must be kept in a valid state. This will ensure that any operations performed by the unit using the liveness table will result in valid output with live registers remaining untouched and only dead registers being altered. The job of keeping the liveness register valid is performed by the compiler or programmer by inserting extra instructions into the stream.

Figure 3 shows a very simple example code fragment. The fragment is analysed by the compiler to produce intermediate instructions, liveness information and a register allocation. This sort of conventional analysis is commonly performed and easily accessible in most modern compiler systems.

By looking at a combination of the liveness information and resulting register allocation, the compiler can deduce which registers are live and dead at any point in the program. This information is used in the code-generation phase to maintain the validity of the liveness table as demonstrated by the output shown in Figure 4. The resulting program contains numerous instances of the *LIVE* and *DEAD* instructions described in Section 2. These instructions act to update the liveness table,

```

00: LIVE R0          ; R0 is live, loaded with a
01: LOADC R0, %1
02: LIVE R1          ; R1 is live, loaded with b
03: LOADC R1, %2
04: LIVE R2          ; R2 is live, loaded with a + b
05: ADD R2, R0, R1
06: DEAD R0          ; R0 is dead
07: MUL R2, R2, R1
08: DEAD R1          ; R1 is dead
09: DEAD R2          ; R2 is dead
0A: LIVE R2          ; R2 is live
0B: LOADC R2, %3
0C: DEAD R2          ; R2 is dead

```

Figure 4. Compiled source code fragment with liveness table instrumentation.

by adding or removing registers from the live and dead sets, and maintain the state as deduced by compiler analysis.

It is vital to note that in this simple example the liveness table is updated after each instruction that alters the current liveness status. Fine grained updating of the table is by no means necessary and, provided that mechanisms that rely on the liveness information understand the consequences, the register could be updated at the boundaries of each basic block. This would significantly reduce the overhead in maintaining the liveness table although the opportunity to use the information could be lessened because of the coarse grained period of validity. Further reduction of the overhead cause by updates to the liveness table can be performed by peephole optimisation. An example of where this could happen is found in instructions, at addresses *0x09* and *0x0A*, of the code in Figure 4. Register *R2* goes dead at the end of one assignment only to become live again directly afterwards. The two instructions that transfer this information to the liveness table can be optimised away given that there are no other instructions between them.

4. Implementation

The concepts of the mutation unit and liveness table are, in isolation, neither particularly interesting or useful. The foundation mechanisms described in Sections 2 and Sections 3 allow us to show how the mutation unit alters the instruction stream as it flows towards the point of execution. There are three main operations carried out by the mutation unit, using the liveness table, in order to implement this alteration of the instruction stream being passed to the execution unit:

- Deletion of instructions marked by the programmer as being non-certain in their execution probability. The *maybe* [8] concept allows a programmer to denote how probable the execution of an instruction is and, at run-time, allow that execution to depend on the result of a random decision. This facilitates the construction of several useful features such as random jump targets and multi-source reductions.
- Insertion of instructions into the stream where either we have previously deleted an instruction, the incoming instruction adds no executable content to the program or there is a potential for the pipeline to stall.
- Alteration of instructions so that their meaning is the same while their register usage and mapping is different. This technique has previously been performed using conventional register renaming [14] technology. However, since we possess more semantic information about the program being executed, in the shape of the liveness table, we can alter the programs

register usage patterns at any point in execution without the need for such costly hardware.

The implementation of instruction insertion and alteration is performed using the concepts of random instruction generation and random register remapping described in Sections 4.1 and 4.2 respectively. These mechanisms, when added to the concept of the *maybe* predicate, form a simple pseudo-code algorithm for the mutation unit which is detailed in the full version of this paper [9].

4.1. Instruction Generation

One way to alter the power profile of the processor between runs is to implement a system of *random instruction generation*, or hardware based *peephole randomisation*, where instructions are added that retain the program meaning while altering the composition of the stream. This technique will result in the executed application having a different instruction mix than the stored application. If this is done at random, an attacker will find it difficult to match the differing resultant power usage with the point in application which could have produced it.

This technique is similar to others [11, 6], which seek to tamper-proof software, in that it introduces extra instructions into the program which is executed. However, unlike tamper-proofing, which obfuscates the program at compile-time to hide the meaning a given code sections, random instruction generation occurs at run-time. Since the source code fed to the processor remains unchanged, it is far harder to filter out the extra instructions from the resulting DPA traces.

4.1.1. Liveness Guided Target Registers

The problem with the technique of inserting instructions at run-time is that the number of potential mutations to the instruction stream that can be considered is restricted by the need to retain valid register contents. That is, we can never issue instructions that alter registers other than the target of the original instruction as doing so would produce incorrect application behaviour. However, in the presence of a liveness table we know the registers that contain valid data and that can therefore be overwritten without consequence. Using the liveness table, it is possible at any point in execution to generate an instruction which targets a dead register and sources either dead or live registers. We can insert this instruction into the stream and have it executed since we know that by doing so we will not overwrite any useful information.

Figure 5 shows how the example code fragment in Figure 4 might look after being operated on by random instruction generation. Considering the code fragment in isolation, we know after each update to the liveness table which registers are live and which are dead. This information enables the hardware to insert an extra instructions, of constrained type, which target only dead registers. Although the meaning of the application remains unchanged, because the overwritten value would never have been used, the power usage and run-time is variable. The variation will depend on the probability of instructions being inserted at a given point and type of inserted instructions inserted. Considering more complex examples, where there are more points at which the instruction stream may be mutated, the lack of determinism in the resultant application will make DPA based attacks much harder to perform.

4.2. Instruction Alteration

4.2.1. Identity Instructions

A simple example of how instructions can be altered to provide the same meaning while having a differing composition is the concept of identity instructions. Provided that caveats about instruc-

```

00: LIVE R0          ; R0 is live, loaded with a
01: LOADC R0, %1
02: LIVE R1          ; R1 is live, loaded with b
03: LOADC R1, %2
04: LIVE R2          ; R2 is live, loaded with a + b
05: ADD R2, R0, R1
06: DEAD R0          ; R0 is dead
07:
08: SUB R0, R1, R2 ; mutation unit adds random instruction
09:
0A: MUL R2, R2, R1
0B: DEAD R1          ; R1 is dead
0C: DEAD R2          ; R2 is dead
0D:
0E: DIV R2, R1, R0 ; mutation unit adds random instruction
0F:
10: LIVE R2          ; R2 is live
11: LOADC R2, %3
12: DEAD R2          ; R2 is dead

```

Figure 5. Compiled example source code fragment with instruction addition.

tion side-effects are adhered to, an instruction can be converted into another, differing instruction which has the same meaning. One such situation where this can be performed is constant/memory substitution. In this case, simple instructions which load constants into registers are translated into memory accesses to a small constant pool. The resulting value in the target register will be the same in both cases but the memory access will generate a much different power profile to the immediate constant load. Even simpler examples of identity mutations logical identities involve bitwise operators such as, for example, *AND*, *OR*, *NOT* and *XOR*.

The implementation of simple transformations can be table driven and is similar to the replacement of single complex operations with a conglomerate of simple operations performed in some software tamper-proofing systems [11]. Although we must be careful not to trigger any side effects, such as flag status, with our identities, Figure 6 demonstrates how simple arithmetic identities can be applied to the example program shown in Figure 4.

Although these identities are perhaps not the most effective when considering their impact on the resultant DPA profile, they provide a good example of what is possible. As long as an identity for a given instruction is available, the processor may decide at random to forward either the identity sequence or the original instruction to the execution unit. Since the attacker has no way of knowing which of these actions was performed, the correlation between the source code program and the executed program will diverge as time progresses.

4.2.2. Liveness Guided Register Remapping

Another way in which DPA can be used to reveal information using a processor power profile is by considering the *bit-flipping* problem. By utilising very fine grained power profiles, it is possible to capture the number of bits flipped when an old value in a register is overwritten by a new one. This style of attack can determine the exact value that has been written to the register. This can, in turn, yield secret information about the application in question.

Hardware units to implement random register renaming [14] have been proposed to combat this problem by effectively making it impossible to know which physical register is being written into when an instruction is executed. These devices result in the power profile for two identical application runs to differ because of the randomisation of bit-flipping as a result of differing register mappings. However, full scale register renaming units are costly and complex in terms of the re-

```

00: LIVE R0          ; R0 is live, loaded with a
01: LOADC R0, %1
02: LIVE R1          ; R1 is live, loaded with b
03: LOADC R1, %2
04: LIVE R2          ; R2 is live, loaded with a + b
05:
06: ADD R2, R0, R1
07: ADD R2, R2, %0   ; mutation unit adds instruction to
08:                  ; implement the identity x + y + 0 = x + y
09:
0A: DEAD R0          ; R0 is dead
0B:
0C: MUL R2, R2, R1
0D: MUL R2, R2, %1   ; mutation unit adds instruction to
0E:                  ; implement the identity x * y * 1 = x * y
0F:
10: DEAD R1          ; R1 is dead
11: DEAD R2          ; R2 is dead
12: LIVE R2          ; R2 is live
13: LOADC R2, %3
14: DEAD R2          ; R2 is dead

```

Figure 6. Compiled source code fragment with identity instruction alteration.

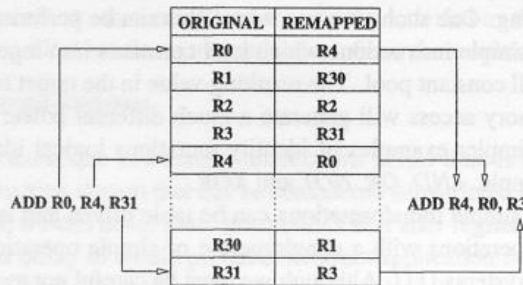


Figure 7. A register remapping table.

sources they require. Furthermore, they are tightly coupled to the host architecture which makes portability and comparison between systems difficult. Because of these limitations, this technique may not be entirely suitable for small, low-power devices such as smart-cards. We can use the liveness table to construct a highly modular alternative solution and negate the need for costly, conventional register renaming hardware. By the addition of a small *register remapping table*, it is possible to harness some advantages of more complex devices while maintaining small silicon size and low-power usage.

The register remapping table is a small array of values which, when indexed by a register number yields the register to which it is mapped. As part of the instruction decode phase, incoming instructions are translated so that any register operands are remapped to new values under control of the table. This process is described pictorially in Figure 7. We know, by looking at the liveness table and by the definition of liveness, that a register deemed to be dead will not be used as a source before it are used as a destination. Using this fact, we can update the register remapping table by taking any dead register and remapping it to any other dead register. The source and destination of the mapping are chosen at random from the set of dead registers so as to introduce non-determinism between application runs. Figure 8 shows how this happens and how the resulting register access pattern will change, from the original shown in Figure 4, as a result of the remapping.

```

00: LIVE R0          ; R0 is live, loaded with a
01: LOADC R0, %1
02: LIVE R1          ; R1 is live, loaded with b
03: LOADC R1, %2
04: LIVE R2          ; R2 is live, loaded with a + b
05: ADD R2, R0, R1
06: DEAD R0          ; R0 is dead
07: MUL R2, R2, R1
08: DEAD R1          ; R1 is dead
09: DEAD R2          ; R2 is dead
0A:
0B:                 ; R1 remapped to R2, R2 remapped to R1
0C:
0E: LIVE R2          ; mutation unit alters to LIVE R1
0B: LOADC R2, %3     ; mutation unit alters to LOADC R1, %3
11: DEAD R2          ; mutation unit alters to DEAD R1

```

Figure 8. Compiled source code fragment with register usage alteration.

As execution progresses and the remapping hardware is continually invoked, the mapping between registers becomes increasingly complex. The different mappings produce vastly differing pairs of old and new values for register writes, a fact that results in different amounts of bit-flipping and associated power use.

5. Results

In order to test the ideas presented in Sections 2 and 3, we implemented the mutation unit as part of a non-deterministic cryptographic co-processing device.

To show the misalignment between source code and DPA power profile, we added power profiling features [5, 19] to the co-processor and performed a DPA attack on it while running the Rijndael [7] block encryption algorithm. The power profile produced by our simulated processor is obviously not real but closely follows the model constructed by researchers performing DPA on real processors [1]. It represents a worst-case situation in that it contains all the information required to mount a DPA attack while omitting any noise that may be introduced by a real processor.

The raw power profiles from experimenting with and without a random issue unit and mutation unit are shown in Figure 9. The graphs show the first few rounds of encryption and demonstrate how the regular patterns of power consumption are disrupted by adding non-determinism elements to a normal processor. Although using the mutation unit to add extra instructions will ultimately use more power, the extra run-time caused by these operations is minimal. This is because the unit only aggressively adds dummy instructions when there are no real instructions available from the previous asynchronous pipeline stage.

From a set of 10000 power profiles, which were generated by performing many encryption operations with the same key and different messages, we mounted a co-variance based DPA attack against the first key-addition operation within the Rijndael algorithm [3]. As part of a more comprehensive results found in the full version of this paper [9], Figure 10 shows a DPA attack against a single bit of the Rijndael key. With no countermeasures, the attack reveals the key bit to be set while using random issue and the mutation unit hides this fact from the attacker, meaning it could be either set or cleared.

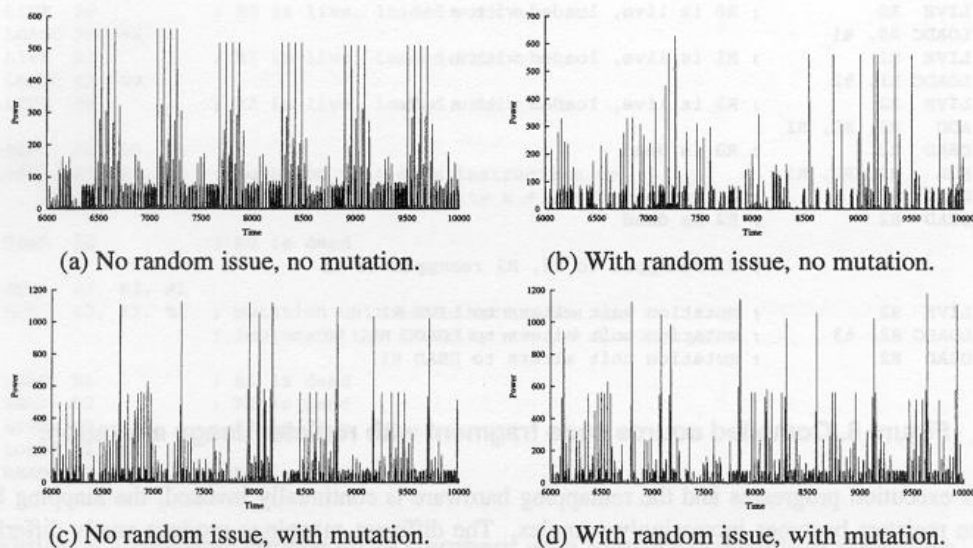


Figure 9. Power profile of processor while running Rijndael encryption algorithm.

6. Conclusions

We have described the augmentation of a conventional processor pipeline that solves a specific high level design problem. With a suitably altered compiler tool-chain, it is clear that that addition of a mutation unit, with an associated liveness table, can increase the level of misalignment between a source program and a DPA profile. This technique draws from existing obfuscation techniques [11, 6] which hide the meaning of application source code. By using the liveness table to allow similar transformations on the run-time instruction stream, the executed program and resultant DPA profile will be obfuscated in a similar manner. The unit compensates for other problems in a non-deterministic pipeline such as where the random issue window is drained of instructions and hence operates in a deterministic manner.

Since the misalignment introduced by this obfuscation will significantly reduce the effectiveness of a DPA attack, the technique is potentially of great value in security applications. Specifically, we expect that vulnerable devices, such as smart-card and embedded applications, will be able to implement this simple system and gain a tangible increase in resistance to DPA attack.

Furthermore, the mutation unit is small and self contained enough to allows its deployment in areas which may otherwise be difficult to augment with non-deterministic processing capabilities. Although the mutation unit requires small changes at the ISA level, the technology is entirely compatible with other non-deterministic processor elements and is general enough to fit into most modern microprocessor design methods.

References

- [1] M-L. Akkar, R. Bevan, P. Dischamp, and D. Moyart. Power Analysis, What Is Now Possible... In *6th International Conference on the Theory and Application of Cryptology and Information Security*. Springer-Verlag, December 2000.
- [2] M-L. Akkar and C. Giraud. An Implementation of DES and AES, Secure against Some Attacks. In *3rd International Workshop on Cryptographic Hardware and Embedded Systems*. Springer-Verlag, May 2001.

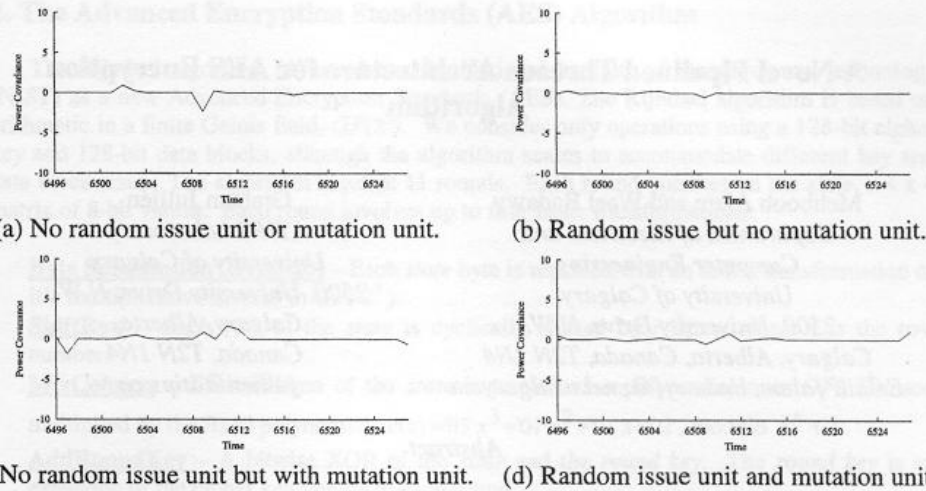


Figure 10. Single bit DPA attack against Rijndael key.

- [3] S. Chari, C. Jutla, J.R. Rao, and P. Rohatgi. A Cautionary Note Regarding Evaluation of AES Candidates on Smart-Cards. In *2nd Advanced Encryption Standard Candidate Conference*, 1999.
- [4] S. Chari, C. Jutla, J.R. Rao, and P. Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In *19th Annual International Cryptology Conference*. Springer-Verlag, August 1999.
- [5] R.Y. Chen, R.M. Owens, M.J. Irwin, and R.S. Bajwa. Validation of an Architectural Level Power Analysis Technique. In *35th ACM/IEEE Design Automation Conference*, June 1998.
- [6] C. Collberg, C. Thomborson, and D. Low. A Taxonomy of Obfuscating Transformations. Technical Report 148, Department of Computer Science, University of Auckland, July 1997.
- [7] J. Daemen and V. Rijmen. AES Proposal: Rijndael, September 1999.
- [8] J. Irwin, H.L. Muller, D. Page, N.P. Smart, and B. Silverman. The MAYBE Instruction. In *Preprint*.
- [9] J. Irwin, D. Page, and N.P. Smart. Instruction Stream Mutation for Non-Deterministic Processors. Technical Report CSTR-01-008, Department of Computer Science, University of Bristol, December 2001.
- [10] P.C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *19th Annual International Cryptology Conference (CRYPTO)*, volume 2139. Springer-Verlag, August 1999.
- [11] M. Mambo, T. Murayama, and E. Okamoto. A Tentative Approach to Constructing Tamper-Resistant Software. In *Proceedings of New Security Paradigms Workshop*, pages 23–33, September 1997.
- [12] M.M. Martin, A. Roth, and C.N. Fischer. Exploiting Dead Value Information. In *International Symposium on Microarchitecture*, pages 125–135, December 1997.
- [13] M.D. May, H.L. Muller, and N.P. Smart. Non-deterministic Processors. In *ACISP 2001*, pages 115–129, 2001.
- [14] M.D. May, H.L. Muller, and N.P. Smart. Random Register Renaming to Foil DPA. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 2162. Springer-Verlag, May 2001.
- [15] T.S. Messerges, E.A. Dabbish, and R.H. Sloan. Investigations of Power Analysis Attacks on Smartcards. In *USENIX Workshop on Smartcard Technology*, pages 151–162, May 1999.
- [16] R. Muth. Register Liveness Analysis of Executable Code. Technical report, Department of Computer Science, The University of Arizona, December 1998.
- [17] A. Srivastava. Link Time Optimization with Translation to Intermediate Program and Following Optimization Techniques Including Program Analysis Code Motion Live Variable Set Generation Order Analysis, Dead Code Elimination and Load Invariant Analysis. Patent US5966539 <http://12.espacenet.com/dips/viewer?PN=US5966539&CY=gb&LG=en&%DB=EPD>, December 1999.
- [18] C. Wang, J. Hill, J. Knight, and J. Davidson. Software Tamper Resistance: Obstructing Static Analysis of Programs. Technical Report CS-2000-12, Department of Computer Science, University of Virginia, December 2000.
- [19] W. Ye, N. Vijaykrishna, M. Kandemir, and M.J. Irwin. The Design and Use of SimplePower: A Cycle-Accurate Energy Estimation Tool. In *37th ACM/IEEE Design Automation Conference*, June 2000.