On Reconfigurable Fabrics and Generic Side-channel Countermeasures

R. Beat[1], P. Grabher[2], D. Page[2], S. Tillich[2], and M. Wójcik[2]

[1] Silicon Basis,
University Gate East,
Park Row, Bristol, BS1 5UB, UK.
rbeat@siliconbasis.com
[2] Department of Computer Science,
University of Bristol,
Merchant Venturers Building,
Woodland Road, Bristol, BS8 1UB, UK.
{grabher,page,tillich,wojcik}@cs.bris.ac.uk

Abstract. The use of field programmable devices in security-critical applications is growing in popularity; in part, this can be attributed to their potential for balancing metrics such as efficiency and algorithm agility. However, in common with non-programmable alternatives, physical attack techniques such as fault and power analysis are a threat. We investigate a family of next-generation field programmable devices, specifically those based on the concept of time sharing, within this context: our results support the premise that extra, inherent flexibility in such devices can offer a range of possibilities for low-overhead, generic countermeasures against physical attack.

1 Introduction

Within the context of countermeasures against physical attack, focusing in particular on power and fault analysis, generality can represent an important consideration. One can view this property as existing in (at least) two dimensions: platform-neutral countermeasures which can be applied to an algorithm implemented on any underlying platform, and algorithmic-neutral countermeasures which can be applied, often in a (semi-) automatic and somewhat transparent manner, to an implementation of any cryptographic algorithm. While the former is exemplified by mathematical and algorithmic augmentation, such as point or scalar blinding [6, Section 5] in Elliptic Curve Cryptography (ECC), the latter represents a more diverse range of techniques. For platforms based on a general-purpose processor, one example is skewing [26, 7] or shuffling [15] the instruction stream: the countermeasure is applied automatically by the processor, irrespective of the purpose of said instruction stream (e.g., whether it represents an implementation of AES or DES). On more hardware-oriented platforms, use of secure logic styles [22, 23] and (ab)use of pipelined logic [20] represent other examples: again the idea is that the approaches are viable no matter what the logic computes, with the only requirement being it is pipelined at all in the latter case.

From here on, we focus on generic countermeasures of an algorithmic-neutral type, using the terms synonymously. Realising such countermeasures can be easier if the underlying platform is more flexible; for example, although general-purpose processors offer flexibility in terms of what they can execute, their fixed design usually requires alteration to support efficient temporal skewing or shuffling. Field Programmable Gate Arrays (FPGAs), and reconfigurable fabrics more generally, therefore represent an interesting option. In particular, their flexibility can allow various generic countermeasures without the associated cost of platform redesign or redeployment (even if it requires reconfiguration). Examples include work by Mentens et al. [16] who harness partial reconfiguration to randomise placement of computational blocks, and Güneysu and Moradi [9] who present a number of approaches including noise generation, clock randomisation and block memory scrambling.

The concept of a Time Multiplexed Field Programmable Gate Array (TMFPGA), see for example [25, 5], is similar to a conventional FPGA wrt. reconfiguration, but resolves significant technological issues (notably logic density) that count against FPGAs in certain use-cases. At a high level, a TMFPGA can be viewed as time sharing resources (such as LUTs and routing blocks) in order to use them more efficiently. Although the underlying technology is less mature than for FPGAs, concrete implementations are emerging: an example is the Tabula ABAX family of 3D Programmable Logic Devices (3PLD), based on the so-called Spacetime Architecture [1].

Soft Gate Array (SGA) [4] is an SRAM-based TMFPGA architecture that aims to overcome the remaining major limitations of TMFPGAs, namely dynamic and static power consumption. One might argue these advantages are well aligned to use-cases where physical attacks are most often an issue (embedded or mobile computing devices, for example). As a result, it is interesting to proactively investigate whether the added flexibility afforded by an SGA can be translated into mechanisms for realising generic countermeasures, with particular focus on fault and power analysis, especially Differential Power Analysis (DPA) [11]. Based on an SGA architecture and associated simulator detailed in Section 2 (including observations on the inherent security of SGAs) we make two novel contributions in this direction. First, in Section 3, and using the results of an existing attack [14] as motivation, we demonstrate how use of an SGA can prevent leakage resulting from glitches and early evaluation. Second, in Section 4 and Section 5, we outline a number of low-overhead alterations to the baseline SGA architecture that permit analogues of existing generic countermeasures to be realised.

2 An overview of SGA-based fabrics

Numerous static features of an SGA implementation are parameterisable. Here we take the most abstract view possible, giving a completely unparameterised description: our assumption is that all parameters are instantiated before man-

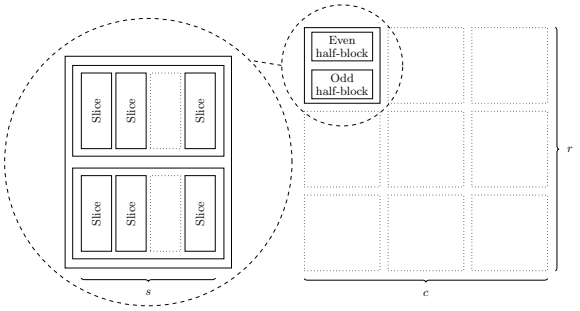


Fig. 1: An SGA logic array is an $(r \times c)$-element collection of logic blocks (right), each of which contains an odd and even half-block (left); each half-block contains s slices.

ufacture to suit an intended use-case (or market) and any restrictions relating to efficiency.

2.1 Resource organisation

At the highest level, an SGA is a collection of resources, organised in a highly regular and hierarchical manner; many resources have FPGA analogues. (e.g., BRAM-style memory blocks). Our focus is almost exclusively the logic array resources: very roughly, these represent the analogue of Configurable Logic Blocks (CLBs) in an FPGA.

Each logic array is an $(r \times c)$-element collection of logic blocks, each of which contains an odd and even half-block; each half-block contains s slices, see Figure 1. Crucially, a set of p one-hot phase clocks is distributed to each slice. The phase clocks drive a system cycle consisting of p phases of evaluation: put another way, they control how a system cycle is time shared into p phases.

2.2 Interconnection network

Resources on an SGA, logic arrays in particular, communicate via an interconnection network. In common with the logic arrays themselves, the interconnect is time shared: the i-th element of a global routing configuration is selected by the phase clocks, controlling how the logic arrays communicate within the i-th phase.

In conventional FPGAs, the analogous interconnect represents a major source of power consumption. Since said consumption varies by the square of the voltage level, significant savings can be achieved by optimising the interconnect. To this end, an SGA interconnect employs a Low Voltage Differential Swing (LVDS) scheme [17]. Rather than a single wire, a differential scheme uses a *pair* of wires: one carries a signal, the other carries the complement of that signal. A sense amplifier, similar in design to those used in memories such as SRAMs, is used

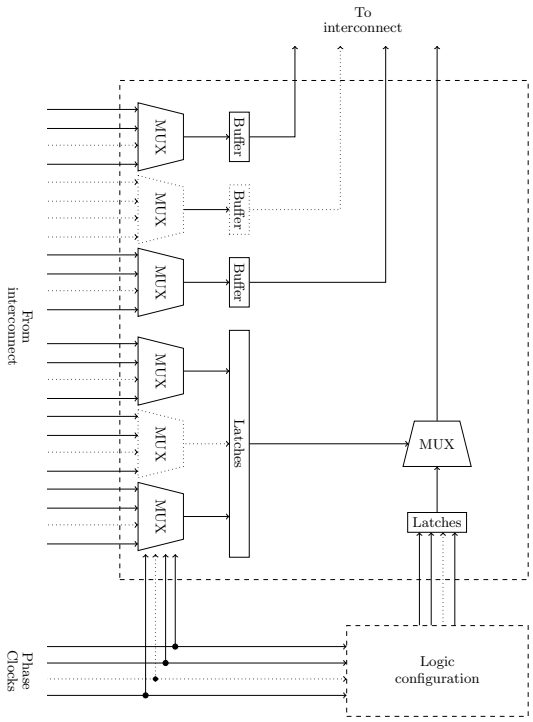


Fig. 2: An SGA slice is controlled by p phase clocks. The active i-th phase clock (bottom left) causes the input multiplexers (left) to select the corresponding input from the interconnect; these select the j-th LUT entry (i.e., the i-th configuration loaded from the attached memory; bottom right), and form an output into the interconnect (top).

to detect small voltage differences on the wire pair; this means two end-points (e.g., logic blocks) can communicate over the interconnect using low voltage levels (about 200 mV), and internally translate the resulting signal into a full voltage level ready for use.

From a security perspective, three features of the interconnect are important:

1. the sum of voltages on the wire pair is (very close to) constant,
2. the current consumed in driving a one or a zero is (very close to) identical, and
3. the overall dynamic power consumption is comparatively very low.

As such, and since many DPA attack strategies target signals transferred over an interconnect (or bus more generally), an SGA already reduces the signal-to-noise ratio and hence attack potency.

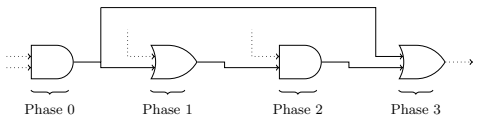


Fig. 3: An illustrative example of SGA operation, using a (contrived) 4-gate combinatorial circuit.

2.3 Slice operation

Although a slice may include specialist components (e.g., carry chains), we focus on their more general role as an n-input, 1-output Boolean function. The principal structure of a slice is depicted in Figure 2. The function computed is determined by the active configuration: in the i-th phase, triggered by the i-th phase clock, the i-th configuration is loaded from a configuration memory (whose physical location is implementation dependent, but which we view as part of the slice).

Each configuration represents a Look-Up Table (LUT) that describes the required function: the j-th element is selected by input from the interconnection network (i.e., from other slices, determined by the routing configuration for the i-th phase) and forms an output sent into the interconnection network (i.e., to other slices). In addition, the slice houses a set of b buffers whose role is to store intermediate results between phases.

As an aside, we stress that division of slices into odd and even half-blocks is an implementation choice: to avoid over complicating our description, and without loss of generality, we ignore this feature. In more detail however, the idea is to support optimisations such as prefetching the configuration for one half-block while the other evaluates. This of course implies additional rules wrt. which slices can act as input and output in a given phase.

2.4 Conceptual example

Use of an SGA demands careful synthesis of a model, written in some Hardware Description Language (HDL) such as Verilog or VHDL, into associated routing and logic configurations. This process is far from trivial, but can be (semi-) automated in much the same way as (and integrated with) a conventional synthesis pipeline.

To clarify various operational aspects, consider the combinatorial circuit in Figure 3 which consists of $m = 4$ gates. Using an FPGA, m separate CLBs might implement the gates and evaluation might require just one cycle; with an SGA, one slice can evaluate the same circuit by dividing evaluation into $p = m$ phases. As such, in the i-th phase the LUT within said slice is specified by the i-th entry in the associated logic configuration: evaluation progresses step by step, with input provided from either previous phases in the same slice (supported by use of one or more buffers), or other slices.

Of course, the parameterisation sets p rather than allowing a selection of $p = m$ at run-time. A case where $p > m$ implies that some phases (wrt. this slice) are essentially NOPs; where $p < m$, the number of phases is insufficient to implement the circuit using one slice. The latter case is of course harder to deal with: typically it requires redesign of the configuration(s) to divide evaluation between more than one slice, or over more than one system cycle.

2.5 A simulated SGA, and experimental method

To provide experimental results for our work, and lacking a physical test device, we developed a VHDL-based cycle-accurate SGA simulator. The simulator is essentially a library of SGA resource blocks which can be parameterised and combined to model said test device. For the purpose of our investigation, we set the fabric parameters to $r = 2$, $c = 2$ and $s = 4$. This gives 4 logic blocks in total, and 32 slices available for use.

To then configure the simulator, and given a HDL model describing the required functionality, we first performed synthesis for a Xilinx Virtex-4 FPGA target (i.e., a target using 4-to-1 LUTs). From the resulting EDIF netlist, we then extracted each LUT configuration and associated interconnections: this information was used to drive an SGA-specific place and route tool which mapped LUT configurations onto slices. A limited amount of manual post-processing was applied to optimise and correct the final placement, routing and configuration.

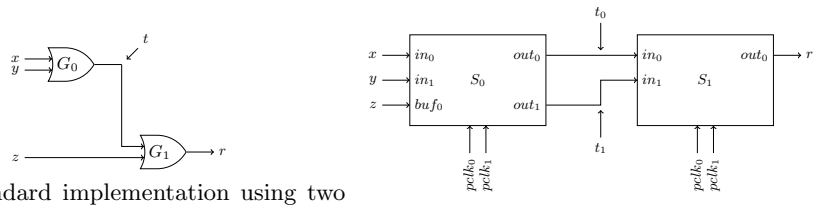
We used a simple power model whereby power consumption scales in proportion to switching behaviour; this metric was extracted from each simulation with an existing framework [10] shown to give reliable estimations within the context of DPA. Each actual DPA attack was performed using the resulting data by applying the OpenSCA toolbox in Matlab [18].

3 Glitches and early evaluation in cryptographic circuits

Mangard et al. [14] describe a DPA attack on an AES S-box with integrated, masking-based countermeasures. The attack succeeds due to transient behaviour, or glitches, on intermediate signals in the circuit. Such glitches typically result from unbalanced paths, and therefore apply to FGPA- and ASIC-based implementations.

An SGA protects against such glitches. One can view evaluation as a p-stage pipeline, wherein each stage is a single gate: the output of each stage is latched before being reused as an input, meaning the same transient behaviour is suppressed. In this section, we demonstrate how said feature can be used to prevent the attack in [14]: by demonstrating the glitch-free behaviour on a motivating example, we conclude that such an approach is a generic countermeasure against similar attacks.

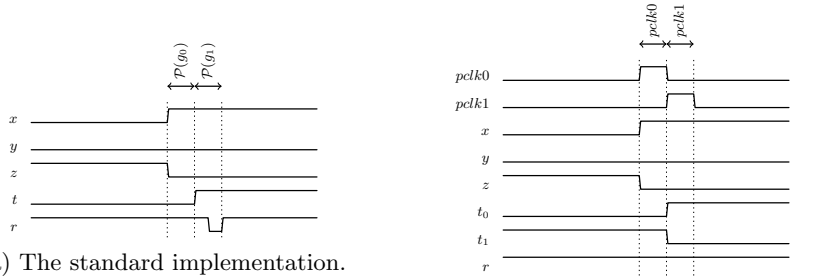
Let $\mathcal{P}(G)$ denote the propagation delay associated with some gate G, and consider the combinatorial circuit in Figure 4: it uses two OR gates, labelled G_0 and G_1, to compute $r = x \vee y \vee z$ (where $t = x \vee y$). The circuit should



(a) A standard implementation using two gates.

(b) An SGA-based implementation using two slices.

Fig. 4: The "hello wORld" combinational circuit used to demonstrate glitches.



(a) The standard implementation.

(b) The SGA-based implementation.

Fig. 5: Evaluation waveforms relating to Figure 4.

output $r = 0$ iff. all input signals are 0, otherwise $r = 1$. However, if one uses an FPGA-like platform, evaluation is modelled by the waveform in Figure 5. Specifically, the transition from where $x = 0$, $y = 0$ and $z = 1$ to $x = 1$, $y = 0$ and $z = 0$ provokes a short glitch where $r = 0$; this is essentially a result of the propagation delay associated with G_0. Using an SGA-based implementation on the other hand, the same glitch is eliminated per the description above.

More generally, an issue which has greatly complicated the development of side-channel resistant logic styles is so-called early evaluation within combinational components [12, 21, 19]. The problem is that the exact point in time when a combinational gate will evaluate its output is dependent on the value and arrival time of its inputs. For example, the standard version of "hello wORld" will evaluate its output r early for certain input transitions. The left-hand part of Figure 6 attempts to visualize the impact of early evaluation on the standard implementation using the number of transitions that occur (i.e., the switching behaviour) during evaluation of the circuit. Two different input transitions ($x = 0$, $y = 0$, $z = 0$ to $x = 1$, $y = 1$, $z = 0$ in the top part, and $x = 0$, $y = 0$, $z = 0$ to $x = 0$, $y = 1$, $z = 1$ on the bottom) are shown. One can see that transitions in the latter case occur earlier as the second OR gate G_1 evaluates early. The right-hand part of Figure 6 shows the transitions for the same functionality

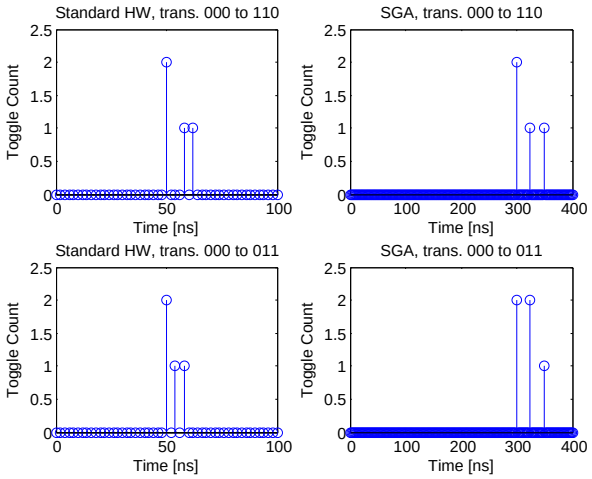
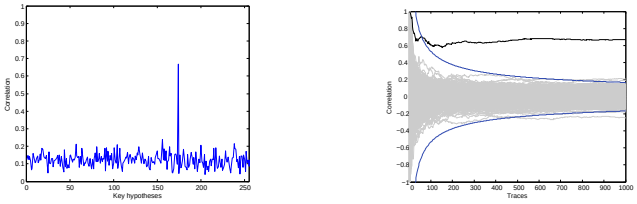


Fig. 6: Switching behaviour of the standard (left) and SGA-based (right) implementations of Figure 4.



(a) The 256 key hypotheses.

(b) Evolution of correlation coefficient.

Fig. 7: Results of a DPA attack on the AES S-box.

implemented on an SGA using two slices, where all transitions occur at fixed points in time.

Note that the number of transitions in the SGA-based implementation is clearly data-dependent (as for *any* CMOS-based hardware). However, standard countermeasures like hiding or masking, which have been often thwarted in the past by the effects of glitching and early evaluation, could potentially be more effective on an SGA which inhibits these effects by design.

Implementation	ρ	Traces required	Used slices	Execution time (phases)
Vanilla AES S-box	0.66	45	20	12
Buffer randomisation	0.27	300	24	12
Phase skewing	0.19	600	20	13
Dummy computation	0.21	650	20	14

Table 1: Results of a DPA attack on the AES S-box: effectiveness of various countermeasures. Note that the number of traces required for a successful attack is an estimate of the security level, and that the number of slices used is out of a possible 32 in total; the unused slices represent additional options for the implementation of countermeasures outlined throughout Section 4.

4 SGA-based countermeasures against power analysis attacks

4.1 Overview

In a power analysis attack, an attacker monitors the power consumption of a target device while it evaluates some function; through analysis of the inputs, outputs and power consumption traces, the attacker hopes to recover some embedded, security-critical information from the target. To illustrate the problem this presents, we mounted a standard attack on a simulated SGA-based implementation of the AES S-box; no countermeasures were employed. Note that this operation alone (rather than the whole cipher for example) is a valid example, since correct prediction of the S-box input from the power consumed during evaluation allows an attacker to recover the entire AES key in a byte-wise fashion. To map the hypothetical S-box values to hypothetical power consumption values, we applied the Hamming-weight model, i.e., we assumed that the power consumption of the SGA is directly related to the number of bits set to one in the data being processed.

Figure 7a illustrates the correlation coefficients computed for all 256 key candidates after processing $1,000$ traces; the correct hypothesis (namely $k = 173$) leads to a significant correlation coefficient of $\rho = 0.66$. Figure 7b illustrates the evolution of correlation as the number of traces increases: incorrect key candidates are plotted in grey colour, while the correct key candidate is highlighted in black. Per [13, Page 148], this allows estimation of the number of traces required for a successful attack (namely ~ 45); the same approach allows estimation of countermeasure efficacy (where relevant).

4.2 Countermeasures

At a high level, and ignoring approaches such as hardware shielding, countermeasures against DPA can be classified as based on either a hiding (breaking the link between execution and traces) or masking (breaking the link between

execution and algorithm) approach. Hiding countermeasures typically attempt to make each trace constant for wrt. all possible values of the security-critical information, or entirely random; in both cases the premise is that a trace is no longer related to said information.

By leaning on existing design features (esp. the high degree of flexibility wrt. when and where computation occurs), an SGA-based fabric can be used to realise various generic countermeasures of these types. Each case requires at most a minor alteration to the base SGA design, and can be described as generic in that application can be (semi-) automated without relying on the functionality (i.e., algorithm) being implemented. Additionally, the countermeasures can often be composed to amplify the security benefits they provide individually. We use the rest of this section to describe both the approaches and results, which are summarised in Table 1.

Buffer randomisation The eight bits of output from the S-box implementation are generated during different phases: to provide a collective final output at some boundary (either phase or system cycle), they need to be selectively buffered before communication to subsequent logic blocks. By design, the buffers are initialised to zero. Thus, under the power model, the Hamming weight of latched data is directly related to power consumption. For that reason, a DPA attack can recover the input with only a small number of traces.

As shown in our results in Table 1, the complexity of such an attack can be significantly increased by *randomly* initialising the buffers instead. Two directions are possible when considering how to realise this approach. First, one might focus on an unaltered SGA and attempt to interleave a PRNG implementation into the S-box using free slices; this would permit the PRNG output to initialise buffers with essentially no overhead. Second, one might alter the SGA to allow each slice to be controlled by a mode flag: in the i-th phase, said flag has the slice either operate as normal, or draw a configuration from some external (local or global) source of randomness (thus randomising computation).

We performed two experiments: the first with reconfiguration of only those slices which directly influence the buffers (additional unused slice configurations were set to zero), and the second with reconfiguration of *all* unused slices with random values. In the former, we increased the number of traces required only marginally to ~ 300; in the latter, the attack failed even given $10,000$ traces. We stress that we artificially aborted the experiment after $10,000$ traces, and that of course generation of additional traces will eventually allow recovery of the target value; the point is that the threshold for success is now so great, the attack should be deemed less viable.

Phase randomisation Imagine selecting an SGA parameterisation where instead of p phases, there are $p' = p + \delta$. One motivation for doing this could be some form of optimisation; for example, more phases in each system cycle might reduce the number of system cycles. Another, more pertinent, motivation is to

include a degree of freedom (governed by δ) wrt. scheduling of computation. Specifically, one might consider:

Phase skewing The idea is to randomise the point in time when a particular step of computation occurs (in our example, when a bit of the S-box output is computed). This is achieved by simply "skipping" δ randomly selected phases, meaning the overall system cycle takes p' phases but a given i-th phase might not be evaluated when expected per the original schedule.

Dummy computation Instead of idle phases as above (which could arguably be detected and eliminated by an attacker), an incremental extension is to have slices compute some dummy (or fake) computation.

In both cases, the randomised control signals required can be obtained relatively easily from existing digital clock managers [9]. Table 1 presents the result of mounting a DPA attack on such implementations. We stress that the experiments use $\delta = 1$ only in order to show the results are consistent with theory: larger choices of δ cause the number of traces required to increase further, trivially achieving a much higher security level (with associated degredation in performance).

With some effort, the same approaches are of course viable on FPGA-like platforms. Crucially however, the fact that phased-based evaluation is inherent on an SGA means the countermeasure can be applied generically; on an FPGA, the same is not true. For example, clock randomisation on an FPGA [9] requires at least some algorithm-specific detail about clock signals and clocking strategy. Additionally, on an SGA randomisation influences delay between the computation of single LUTs, and thus makes the countermeasure very fine-grained. The same is not true for FPGAs, where the same delays will relate to computation between two consecutive flip-flops with combinational logic between them: usually this represents a more course-grained approach.

A more aggressive approach still, which we defer to further work, would be a phase-oriented analogue to instruction shuffling; the idea would be to *reorder* phases (while retaining dependencies) instead of skewing them in time. Versus skewing per the description above, similar security benefits result *and* potential improvements wrt. efficiency are possible: if the phase dependencies allow, δ could be small (even zero) while producing the same security benefit. High-level, algorithm-specific implementations of this concept in hardware are known [27]; realising an algorithm-neutral version on an SCA seems difficult as the result of managing dependencies between phases. This is, for example, more difficult than the case of instruction dependencies in software [15].

Complimentary computation Approaches to ensuring constant power consumption during evaluation of some functionality can be considered at a variety of levels. For example, at a low-level the concept of specialist logic styles [22, 23] can be considered; at a higher level options such as MUTE-AES [2] are possible. In the latter, the idea is to have two processors execute the same operation in lock-step, but ensure one computes with intermediate data that is the complement of the other: in essence, power consumption is balanced at each step of

computation. However, realistic use of the concept needs to consider at least two criticisms, namely

1. construction of suitable complementary functionality seems hard to generalise to all high-level algorithms, and
2. a careful approach to synchronisation of steps, and the problem of early evaluation within those steps, is required.

Per Section 2.2, an SGA interconnect uses an LVDS scheme: this means signals are (and hence communication is) balanced by design. In addition, the availability of original and complement signals means computation can be also balanced with relatively little overhead: one produces a complementary LUT for each original LUT, and places the resulting slices so both evaluate in the same phase (using appropriate inputs from the interconnect). Crucially, the approach is generic and no alteration to the SGA architecture is required: only (semi-) automatable effort at design-time during synthesis and place and route is needed.

We note that somewhat analogous techniques exist for ASIC [24, 8] and FPGA [3] platforms. For an SGA however, one can reasonably expect less overhead wrt. routing (by virtue of existing interconnect design), and more flexibility wrt. any area constraints. Specifically, duplication of resources (per the FPGA-based solution) to ensure balanced power consumption can easily hit a limit: with m LUTs in the before, at least $2m$ are required afterwards. If an FPGA has less than $2m$ LUTs, the approach is simply not viable. With SGA however, time sharing allows one to "spread" those $2m$ LUTs over the same number of slices but more phases, effectively making the trade-off between security and time rather than security and area.

5 SGA-based countermeasures against fault analysis attacks

5.1 Overview

In contrast to side-channel attacks such as power analysis, where the attacker is largely passive, fault attacks allow active manipulation of the target: after any initial depackaging, invasive transient and/or permanent faults are injected during evaluation. Attacks are mounted wrt. a fault model which constrains the viability of faults; for example, corrupting some unknown bit of internal state is viable, but replacing a specific word with some chosen value is less so. Through analysis of inputs and corresponding correct and faulty outputs, the attacker hopes to recover some embedded, security-critical information from the target.

5.2 Countermeasures

As above, and again ignoring approaches such as hardware shielding, one can identify several broad categories of countermeasures against fault attack: hiding (breaking the link between execution and faults), duplication-based (where

computation is performed more than once, then checked to ensure a consistent result), checksum-based (where auxiliary, but not duplicate computation is checked to ensure a consistent result).

Redundant computation Classical fault detection mechanisms are based on redundant computation, i.e., computing the same result twice and checking for consistency. The assumption is that injecting the same fault twice is difficult, hence the countermeasure provides a generic form of detection. The overhead is, however, significant. One approach is to physically duplicate the target and perform the computations in parallel, another is to keep one implementation and use it twice in sequence; the former doubles the area requirement, the latter doubles the time requirement.

An SGA permits a compromise between these two extremes. Usually, a given slice (or slices) will include unused configurations (or "bubbles") due to dependencies in computation. Instead of leaving these resources idle, essentially representing NOPs, they could be used to implement redundant computation to detect faults: the idea would essentially be to interweave the original and redundant functionality, hoping for closer to full utilisation overall. Moreover, using a similar argument as in Section 4.2, duplication of hardware resources can hit a limit on an FPGA, however, on an SGA time sharing permits to "spread" redundant computation over the same number of LUTs but more phases.

Spatial randomisation Some fault attacks require the ability to inject faults at a specific physical location (for example, in a specific register) on the target device. A countermeasure that capitalises on this requirement is the concept of spatial randomisation: as often as every use of the target, the aim is to randomise placement of constituent functional blocks (e.g., LUTs or BRAMs) meaning a fault cannot be consistently injected at the required location. Mentens et al. [16] use partial reconfiguration to apply this concept on FPGAs. Arguably, two disadvantages result:

1. the approach is non-generic since functional blocks (and routing) must be carefully extracted, and
2. the speed at which reconfiguration is possible limits how often the location can be randomised (and hence the level of security offered).

In contrast, an SGA implementation of the same concept can be generic by virtue of phased-based evaluation: provided the routing configuration is updated in line, the behaviour of some slice can be relocated to another fairly easily. The simplest form possible is interchanging two slices, S_0 and S_1 say, i.e., swapping their functionality which can be described by F_0 and F_1. The idea is to set the logic configuration of both S_0 *and* S_1 to include both F_0 *and* F_1, each consuming half the entries. Now some global, random signal r is passed to S_0 and S_1, each of which uses it to select which half of the configuration to evaluate: S_0 evaluates F_r and S_1 evaluates F_{1-r} meaning, for example, that if $r = 0$ we have S_0 evaluate F_0 and S_1 evaluate F_1 and visa versa. This of course offers only a limited form of

spatial randomisation, *but* the overhead is minimal (assuming a change in higher-level SGA control). One can imagine much more complex forms of relocation *if* the associated overhead is viable, but in either case one could expect the cost to be vastly lower than partial reconfiguration using an FPGA.

6 Conclusion

In this paper, we have demonstrated the advantages that extra flexibility in the design of field programmable logic can afford wrt. security. With a focus on an SGA architecture, but the concept of time sharing more generally, we showed how a range of countermeasures against fault and power analysis attacks can be realised in a low-overhead manner (versus alternatives such as FPGAs).

On one hand, simulated results can only go so far: as with most aspects of physical security, detail relating to concrete implementation can be very important. On the other hand, similar devices are already gaining traction (cf. the Tabula ABAX family). With this in mind, proactive rather than reactive (esp. once deployed) investigation of such topics can act as an important design guide. For example, results in Section 4 and Section 5 demonstrate that only minor alterations to a baseline SGA architecture can yield tangible benefits. Put another way, treating security (in this case against fault and power analysis, but more generally also) as a first-class design metric now could allow more satisfactory use in security-critical applications in the future.

Based on the initial potential illustrated here, one can identify (at least) three areas of further work:

1. use of a more accurate power model to mitigate the use of simulation and improve relevance to physical test devices,
2. study of a full AES implementation, and
3. answer some questions wrt. an SGA-specific tool-chain, in particular whether it is feasibility to realise the generic countermeasures in a fully automatic way.

Acknowledgements

The work described in this paper has been supported in part by EPSRC grant EP/H001689/1.

References

1. Tabula Spacetime Architecture. Technical report, Tabula Inc., 2010.
2. J.A. Ambrose, S. Parameswaran, and A. Ignjatovic. A Multiprocessor Architecture to prevent Power Analysis based Side Channel Attack of the AES Algorithm. In *ICCAD*, pages 678–684, 2008.
3. K. Baddam and M. Zwolinski. Divided Backend Duplication Methodology for Balanced Dual Rail Routing. In *CHES*, pages 396–410. LNCS 5154, 2008.

4. R. Beat. Programmable Logic Fabric. US Patent Application 2011/0031999 A1, February 2011.
5. N.B. Bhat, K. Chaudhary, and E.S. Kuh. Performance-Oriented Fully Routable Dynamic Architecture for a Field Programmable Logic Device. Technical Report UCB/ERL M93/42, EECS Department, University of California, Berkeley, 1993.
6. J.-S. Coron. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In *CHES*, pages 292–302. LNCS 1717, 1999.
7. J.-S. Coron and I. Kizhvatov. An Efficient Method for Random Delay Generation in Embedded Software. In *CHES*, pages 156–170. LNCS 5747, 2009.
8. S. Guilley, P. Hoogvorst, Y. Mathieu, and R. Pacalet. The "Backend Duplication" Method. In *CHES*, pages 383–397. LNCS 3659, 2005.
9. T. Güneysu and A. Moradi. Generic Side-Channel Countermeasures for Reconfigurable Devices. In *CHES*, pages 33–48. LNCS 6917, 2011.
10. M. Kirschbaum and T. Popp. Evaluation of Power Estimation Methods Based on Logic Simulations. In *AUSTROCHIP*, pages 45–51, 2007.
11. P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *CRYPTO*, pages 388–397. LNCS 1666, 1999.
12. K.J. Kulikowski, M.G. Karpovsky, and A. Taubin. Power Attacks on Secure Hardware Based on Early Propagation of Data. In *IOLTS*, pages 131–138, 2006.
13. S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer-Verlag, 2007.
14. S. Mangard, N. Pramstaller, and E. Oswald. Successfully Attacking Masked AES Hardware Implementations. In *CHES*, pages 157–171. LNCS 3659, 2005.
15. D. May, H.L. Muller, and N.P. Smart. Non-deterministic Processors. In *ACISP*, pages 115–129. LNCS 2119, 2001.
16. N. Mentens, B. Gierlichs, and I. Verbauwhede. Power and Fault Analysis Resistance in Hardware through Dynamic Reconfiguration. In *CHES*, pages 346–362. LNCS 5154, 2008.
17. National Semiconductor. LVDS Owner's Manual, 2008.
18. E. Oswald. OpenSCA: An open source toolbox for Matlab.
19. T. Popp, M. Kirschbaum, T. Zefferer, and S. Mangard. Evaluation of the Masked Logic Style MDPL on a Prototype Chip. In *CHES*, pages 81–94. LNCS 4727, 2007.
20. F.-X. Standaert, S. Berna Örs, and B. Preneel. Power Analysis of an FPGA: Implementation of Rijndael: Is Pipelining a DPA Countermeasure? In *CHES*, pages 30–44. LNCS 3156, 2004.
21. D. Suzuki and M. Saeki. Security Evaluation of DPA Countermeasures Using Dual-Rail Pre-charge Logic Style. In *CHES*, pages 255–269. LNCS 4249, 2006.
22. K. Tiri, M. Akmal, and I. Verbauwhede. A Dynamic and Differential CMOS Logic with Signal Independent Power Consumption to Withstand Differential Power Analysis on Smart Cards. In *ESSCIRC*, pages 403–406, 2002.
23. K. Tiri and I. Verbauwhede. A Logic Level Design Methodology for a Secure DPA Resistant ASIC or FPGA Implementation. In *DATE*, pages 246–251, 2004.
24. K. Tiri and I. Verbauwhede. Place and Route for Secure Standard Cell Design. In *CARDIS*, pages 143–158, 2004.
25. S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A time-multiplexed FPGA. In *FPGAs for Custom Computing Machines*, pages 22–28, 1997.
26. M. Tunstall and O. Benoit. Efficient Use of Random Delays in Embedded Software. In *WISTP*, pages 27–38. LNCS 4462, 2007.
27. W. Yang, J. Xu, Y. Yan, and K. Liu. Research on Time Randomization of AES against Differential Power Analysis. In *ISCID*, pages 536–539, 2009.