An Exploration of Mechanisms for Dynamic Cryptographic Instruction Set Extension

Philipp Grabher[1], Johann Großschädl[2], Simon Hoerder[1], Kimmo Järvinen[3], Dan Page[1], Stefan Tillich[1], and Marcin Wójcik[1]

[1] University of Bristol, Department of Computer Science,
Merchant Venturers Building, Woodland Road, Bristol, BS8 1UB, UK
{grabher,hoerder,page,tillich,wojcik}@cs.bris.ac.uk

[2] University of Luxembourg, FSTC, CSC Research Unit, LACS,
6, rue Richard Coudenhove-Kalergi, L–1359 Luxembourg, Luxembourg
johann.groszschaedl@uni.lu

[3] Aalto University, Department of Information and Computer Science,
P.O. Box 15400, FI–00076 Aalto, Finland
kimmo.jarvinen@aalto.fi

Abstract. Instruction Set Extensions (ISEs) supplement a host processor with special-purpose, typically fixed-function hardware components and instructions to utilize them. For cryptographic use-cases, this can be very effective due to the demand for non-standard or niche operations that are not supported by general-purpose architectures. However, one disadvantage of fixed-function ISEs is inflexibility, contradicting a need for "algorithm agility." This paper explores a new approach, namely the provision of re-configurable mechanisms to support dynamic (run-time changeable) ISEs. Our results, obtained using an FPGA-based LEON3 prototype, show that this approach provides a flexible general-purpose platform for cryptographic ISEs with all known advantages of previous work, but relies on careful analysis of the associated security issues.

Keywords: FPGA, embedded processor, instruction set extension.

1 Introduction

Cryptographic kernels could be described as the archetype target for Instruction Set Extensions (ISEs) [31]. Starting with a general-purpose host processor, the idea is to specify an ideally minimal set of (more) special-purpose instructions [25]. By carefully integrating instructions, plus any tightly coupled hardware to support their execution, the goal is more effective implementation of the kernel in question (e.g., with respect to efficiency, memory footprint, or security). This approach can be ideal for cryptography where performance bottlenecks often relate to non-standard or niche operations, and can easily be resolved using a targeted ISE. There exists a wealth of related work to support this premise, see e.g., [10, 17, 31]. Even when focused on one kernel such as AES, said work spans academic results and evaluation on platforms such as the LEON3, through to commercialisation in workstation-class Intel processors via AES-NI [34].

However, at least two valid counterarguments can be considered. First, even though ISEs are often presented theoretically as "non-invasive," their concrete realisation may still be problematic. For example, one can imagine the difficulty of altering incumbent processor designs contributed to the fact that Intel's AES-NI appeared long after suggested by initial work in this area [31]; issues of re-design, re-verification, and re-deployment are scientifically non-trivial and potentially *very* costly. Second, one has to consider the problems of utilisation and flexibility. One aspect is ensuring the cost of design and implementation is worthwhile, another is ensuring ISEs are useful to as many kernels as possible (i.e., making an ISE flexible enough to cater for the future). Cryptography, in particular, has a vested interest in the latter: if an (inflexible) special-purpose ISE for a kernel is deployed and the kernel is then broken, the ISE, associated hardware and sunk design cost subsequently represent useless overhead.

With all these counterarguments in mind, it is interesting to consider how an ISE-based approach, in the most general sense, might be accommodated by next-generation processors. In particular, how might next-generation general-purpose processor designs support dynamic (i.e., changeable at run-time under control of the program) instruction set extension and execution. In both embedded and non-embedded contexts, we already have some answers: focusing on functional units for example, both the Stretch S6000 and ARM-based Triscend A7 include similar concepts which can be dynamically re-configured at run-time, and the new Intel Atom E600C (or "Stellarton") series includes an coarsely integrated (i.e., coprocessor-like) Altera FPGA.

While re-configurable devices such as FPGAs have redefined the traditional roles of hardware and software, re-configurable general-purpose processors are now also a reality; the question is, how does this direction match cryptographic use-cases? This is far from a new topic, but we make progress via four main contributions; we stress that our focus is at the level of micro-architecture and instruction sets, rather than the device level. First we survey strands of related work that support dynamic instruction set extension and execution; second we present PREON (short for Partially Reconfigurable LEON), a novel LEON3-based prototype which includes two such mechanisms; third we evaluate a range of cryptographic primitives on said prototype, demonstrating that it provides a general-purpose platform capable of supporting many existing ISE proposals and specifying some novel additions (e.g., for the two hash functions Skein and JH); and finally, we extend previous security analysis to highlight several issues that require resolution in order to support cryptographic workloads.

2 Background and Analysis

In this section, we present a limited survey of mechanisms that support the concept of dynamic instruction set extension and execution in different ways. Each mechanism has a rich lineage within the field of computer architecture, and is sufficiently mature to exist in production (and in some cases embedded) processor designs [1]. Our overview spans implicit (i.e., invisible to the program) and explicit (i.e., under control of the program) mechanisms; in the latter case we expect that, in addition to the hardware components, there will be generic need for inclusion of management and invocation instruction sets.

Re-Configurable Computation Fabric. The idea is to extend the computational logic (e.g., the ALU) such that, instead of only computing operations which are fixed at design-time, it can be re-configured at run-time. A fairly current and comprehensive overview of the design space is given by Dales [6, Sect. 2.3] and Amano [1], the latter including examples of commercialisation. A more limited list of instances includes

- partly re-configurable functional units, for example CryptoManiac [37] and PipeRench [29],
- tightly integrated run-time re-configurable logic, for example the Triscend A7, Stretch S6000, and Infineon CARMEL, and
- coarsely integrated run-time re-configurable logic, for example Intel's Atom E600C with integrated Altera FPGA.

The choice of fabric can imply extra design constraints whose relevance depends on the context. For example, an FPGA-based fabric could limit the maximum clock frequency, but in an embedded context this may not be of primary concern unless the cost of a mixed technology approach is prohibitive.

Advanced Mechanisms for Instruction Delivery. The idea is to extend the fetch unit so that the mechanism for instruction delivery is (partly) controlled by the program being executed. There is a huge range of related concepts and concrete implementations; a non-exhaustive list includes mechanisms for

- instruction fusion [22], for example load-modify operations in Intel's Core2 micro-architecture and multiply-accumulate in DSP-like (or embedded) processors, allowing composite micro-operations [8] to be specified by a single ISA-level instruction,
- macro-like [32] translation, for example the "sequencer unit" within the IBM RISC Single Chip (RSC) and PowerPC [24] processor, and the ARM Jazelle framework for acceleration of Java programs,
- processor-controlled cache-like structures, for example the trace cache and loop buffer designs within the Core2 and NetBurst micro-architectures, and
- user-controller memory structures such as the register-based buffer of Hines et al. [13], and the now well-studied ideas of scratch-pad memories [2] and non-transparent caches [23].

3 PREON: A LEON3-based Experimental Prototype

In this section, we introduce PREON, a prototype implementation of selected mechanisms surveyed in Section 2. As a starting point we used the LEON3, an open-source implementation of a 32-bit SPARC V8 compliant processor core developed by Gaisler Research AB. We altered the 7-stage LEON3 pipeline as described in detail below, and equipped it with Harvard-style instruction and data caches (or I-cache and D-cache), each 4 kB in size.

The PREON prototype was synthesised to the SASEBO-GII evaluation platform, which we used to produce the experimental results given in Section 4; the processor core itself required 2338 slices of the Xilinx Virtex-5 FPGA (model XC5VLX50-1FF324). The PREON core is clocked at 24 MHz, although we stress that this is a limit imposed by the SASEBO on-board clock.

3.1 Re-configurable Fabric

The first addition is a re-configurable fabric, tightly integrated with the execution unit. We view the fabric essentially as an FPGA, but clearly less general alternatives are viable and potentially preferable in certain contexts. Inclusion of this mechanism is mainly motivated by the goal of improved computational throughput per-instruction (rather than, say, instruction throughput), without compromising flexibility; the fabric can be re-configured to efficiently compute what the general-purpose processor cannot. At least two further benefits result: first, the mechanism reduces communication latency versus a coarsely integrated alternative (e.g., co-processor), and second, it permits the sharing of resources between the processor and fabric (e.g., storage such as registers).

Design and Programming Interface. Use of the fabric by a program executing on PREON is achieved through a single extra instruction named `fabric`. When executed, the instruction has the semantics

$$GPR[dst] = f(GPR[src_1], GPR[src_2], imm)$$

where f denotes a single-cycle functionality provided by the resident configuration: the 32-bit content of the registers src_1 and src_2 (with an 8-bit immediate value imm) is presented to the fabric as input, and the output (according to the current configuration) is stored in register dst. Essentially, the fabric acts as a replacement for the ALU. One can imagine a few alternative models, but this approach allows the fabric to use imm as a means of first specifying any sub-operation (e.g., to house more than one operation on the re-configurable fabric and select between them), and second supplying any immediate data.

PREON allows a configuration resident in the fabric to maintain short term state, e.g., house a register. On one hand, this is arguably outside the traditional remit of ISEs. On the other hand, it allows a high degree of flexibility since for example, multi-input and/or multi-output operations can be supported. In the former case, the fabric is configured to include an “operand fetch” operation which takes the two operands and stores them internally; the stored operands can then be combined with two more operands in a “normal” operation. This feature could be used to overcome the restrictions of the SPARC V8 3-address instruction format without invasive alteration of the micro- or instruction set architecture. Likewise, multi-cycle operations are possible: the fabric is configured to include a “bubble” or NOP-style operation that can be used to absorb cycles while computing the required output. Some other approaches to increasing the input or output bandwidth are possible, e.g., those proposed by Kluter et al. in [19], but our aim is to limit the amount of extra bespoke hardware required.

Implementation. In practice, one would expect the re-configurable fabric to be implemented using a different technology than the processor core. However, the PREON prototype demands partial re-configuration of a host FPGA representing *both* components. Per [38], this requires partitioning at the top-level: we had to divide our design into a static part (the LEON3 processor) and a dynamic part (the re-configurable fabric). In addition, the SASEBO-GII includes 2 Mbit of on-board SRAM memory which we use to store partial bit-streams. A Xilinx XPS HWICAP core, altered with a wrapper for the LEON3 AMBA bus, is used to interface with the FPGA's Internal Configuration Access Port (ICAP). The size of the SRAM, plus the 700 slices reserved on the FPGA for the dynamic part, set an artificial upper limit on the length of the bit-stream, and hence also limit the complexity of configurations used by PREON.

Integration with the LEON3 core is relatively easy: we simply use the fabric rather than the ALU for `fabric` instructions, abusing the 8-bit Address Space Identifier (ASI) register to supply *imm*. Re-configuration of the fabric is done in software by the processor core; essentially, this boils down to a `memcpy`-style transfer of content from the SRAM into the fabric via the ICAP interface. In theory, it is possible to partition the fabric and allow multiple configurations to be resident at the same time; Dales [6, Section 5] outlines various techniques to manage this, but for simplicity we consider a single configuration only.

3.2 Instruction Register File

Since the LEON3 can already be extended with a scratch-pad for instructions (the so-called ILRAM), it is reasonable to question the novelty of our second addition. Crucially however, the concept of an Instruction Register File (IRF), as described for example by Hines et al. [13], captures program fragments whose form relates to basic blocks rather than functions. In short, we suggest that an appropriate IRF design can provide macro-like cryptographic ISEs: the idea is to record short instruction sequences on-chip, then later replay them from the IRF rather than main memory. Rather than "extended" computational ability during execution, the IS"E" here is an "expansion" from a single instruction to a semantically richer *straight-line* instruction sequence.

Inclusion of the mechanism is motivated by two main goals: it should reduce off-chip memory access (which implies lower power consumption), and provide low-latency and deterministic fetch behaviour (both without the physical overhead of an instruction cache). Our premise is that cryptographic use-cases are ideally suited to take advantage of these features, and also benefit from them as a result, for example, of a need to avoid cache-based attacks.

Design and Programming Interface. Our realisation of the IRF concept uses a few small buffers into which instructions are placed and retrieved. Let $B[i][j]$ denote the j-th entry in the i-th buffer where $0 \leq j < n$ and $0 \leq i < m$, i.e., there are m buffers, each of n elements. Let $C[i]$ denote the number of valid instructions currently held in the i-th buffer, meaning that $0 \leq C[i] < n$ for all i. Three additional instructions are used to control these structures:

```
! AES T-table block #1
! input  : packed AES state in %o0 to %o3
!          packed AES round key in %o4 to %o7
!          T-table base addresses in %i3 to %i6
! output : equivalent of %l4 = T-table0[ ( %o0 >>  0 )& 0xFF ] ^
!                              T-table1[ ( %o1 >>  8 )& 0xFF ] ^
!                              T-table2[ ( %o2 >> 16 )& 0xFF ] ^
!                              T-table3[ ( %o3 >> 24 )& 0xFF ] ^ %o4;
record %g0,   0, %g0
srl    %o0,  22, %l4 ; and %l4, 1020, %l4 ; ld [%l4 + %i3], %l4
srl    %o1,  14, %l5 ; and %l5, 1020, %l5 ; ld [%l5 + %i4], %l5
srl    %o2,   6, %l6 ; and %l6, 1020, %l6 ; ld [%l6 + %i5], %l6
sll    %o3,   2, %l7 ; and %l7, 1020, %l7 ; ld [%l7 + %i6], %l7
xor    %l4, %l5, %l4 ; xor %l4,  %l6, %l4
xor    %l4, %l7, %l4 ; xor %l4,  %o4, %l4
stop   %g0,   0, %g0
! AES T-table block #2
...
! AES T-table block #3
...
! AES T-table block #4
...
! AES round
play   %g0,  0, %g0 ! playback T-table block #1 once
play   %g0, 32, %g0 ! playback T-table block #2 once
play   %g0, 64, %g0 ! playback T-table block #3 once
play   %g0, 96, %g0 ! playback T-table block #4 once
...
```

Fig. 1. A sketched example of IRF use: each "block" of a T-table based AES implementation (including key addition) is recorded as a macro into an IRF buffer, then later played back and expanded to form a (non-final) AES round.

- `record` takes an immediate operand i (which specifies a buffer number) and places the fetch unit into recording mode. This acts to redirect instructions into the i-th buffer rather than the pipeline:
  1. initially set $C[i] = 0$,
  2. for each instruction received from the fetch unit, if the instruction is `stop` then act appropriately, otherwise store it to $B[i][C[i]]$, and update $C[i] \leftarrow C[i] + 1 \pmod{n}$.
- `stop` returns the fetch unit to normal mode, redirecting the instruction stream into the pipeline again.
- `play` takes two immediate operands i and c (specifying a buffer number and playback count) and places the fetch unit into playback mode. This acts to inject instructions into the pipeline from the i-th buffer, rather than the fetch unit, c times
  1. freeze the program counter,
  2. inject each j-th instruction from $B[i][j]$, for $0 \leq j < C[i]$, into the pipeline, repeating the process c times then
  3. put the fetch unit back into normal mode, and resume execution from the frozen program counter.

We note that it may be of value to store pre-decoded content in the buffer (like in a trace cache), but defer this topic for further work.

Implementation. Implementation of the IRF in PREON is relatively simple: we follow a conventional (data oriented) register file design, using flip-flops to store content. This structure is controlled by a state machine in the existing LEON3 fetch unit, which applies appropriate operating rules according to the description above. The result contrasts with the more heavy-weight, general-purpose memory approach of an ILRAM in both form and function. More precisely, an ILRAM-based approach implies a larger storage capacity and more involved interface (i.e., memory transaction). Additionally, executing an instruction sequence in an ILRAM demands a branch into and back from said sequence; even if each instruction is retrieved with low latency, the additional branches cause a significant overhead for short sequences. The same is not true of IRF use as there is just a single cycle overhead relating to each `playback`.

Although the description allows general parametrisation (perhaps restricting m and n to powers-of-two), concrete parameters must be selected before use. In theory, the parameters could be selected at run-time using special configuration instructions; this would allow for a high degree of flexibility at relatively marginal cost. However, for simplicity, our PREON prototype currently caters for cases where $m \cdot n = 64$, e.g., $m = 4$, $n = 16$, fixed at design-time after analysis of the associated trade-off. Although larger m or n may improve our results below in theory, our choice tries to balance this against practicality; for example, a total on-chip storage of $64 \cdot 4 = 256$ B matches the capacity of the SSE register file in x86-64 processors.

4 Evaluation of Cryptographic Workloads

4.1 Re-configurable Fabric

Limited evaluations of cryptographic kernels, executed via a similar mechanism with respect to the re-configurable fabric, exist; for example, Dales [6, Section 4.3.2.3] details some experiments with Twofish. In the following, we extend this to include a broader set of modern kernels. Table 1 shows a range of empirical results produced using our prototype PREON implementation. Each result compares a C implementation to a fabric-supported[4] alternative, both with inline assembly statements where appropriate (e.g., to invoke the fabric, or to access SPARC-specific functionality).

Each configuration is designed to match the critical path of the processor; no configuration extends the existing critical path, except the $\mathbb{F}_{3^m}$ multiplier. The execution times (i.e., cycle counts) are averaged over a number of randomised inputs. Although few kernels have data dependent control-flow, this approach takes into account the behaviour of both data and instruction cache. Also note that techniques for automatic identification of configurations, as in [25], seem applicable, but we defer investigation of this topic to future work.

We use the subsections below to discuss each implementation, and conclude with a summary of the results.

[4] To satisfy space restrictions we omit the formal description of each ISE, opting to include a complete description in a full version of this paper.

Table 1. Experimental results comparing the performance of various cryptographic kernels without and with support of ISEs provided by the re-configurable fabric. Note that the static footprint includes instructions *and* any major static data (e.g., T-tables and expanded key schedule), and that initialisation of the fabric is not included in the total number of cycles, rather as a column in the table.

	WITHOUT ISE		WITH ISE			
	Performance (cycles)	Static footprint (bytes)	Performance (cycles)	Static footprint (bytes)	ISE area (slices)	ISE re-config. (µs)
AES						
AES-128 encryption	1281	6068	463	412	115	177
SHA2/SHA3						
SHA-256, 4096-bit message	45241	3304	30528	2492	48	118
JH-256, 4096-bit message	6584962	2052	976372	2116	26	59
Skein-512-256, 4096-bit message	332739	8152	117123	6340	319	470
Grøstl-256, 4096-bit message	258389	16248	152169	1980	112	177
MULTIPLICATION IN $\mathbb{Z}_N^*$						
1024-bit multiplication	86460	768	25148	428	321	590
MULTIPLICATION IN $\mathbb{F}_{2^{233}}$						
School-book	30864	548	2290	428	170	295
Width-4 comb	14900	908	12908	724	44	118
MULTIPLICATION IN $\mathbb{F}_{3^{337}}$						
School-book, bit-sliced	163340	1504	6985	828	690	1062
School-book, bit-serial	445670	1616	11898	420	343	590
Width-4 comb, bit-sliced	82940	4100	56380	3596	70	118
Width-4 comb, bit-serial	247281	8484	40213	2930	54	118

AES. Tillich et al. [31] proposed a set of ISEs that permit efficient implementation of AES on 32-bit architectures, focusing on SPARC V8-based LEON2 in particular. We adopt two ISE classes [31, pp. 275–276], namely `sbox4s` (plus `sbox4r`) and `mixcol4s` (and inverses), and compare them with a T-tables based implementation in software. We note that acceleration of bit-sliced implementations of AES following [10] is viable, but do not investigate this further.

SHA-2/SHA-3. In terms of ISE, SHA-2, focusing on SHA-256 in particular, has been paid relatively little attention. Juliato et al. offer in [17] an exception and explore different hardware/software approaches that include ISEs for rotation and the Ch and Maj functions; we follow their approach fairly directly.

Regarding SHA-3, we stress that we do *not* aim to compare the five finalists directly, but rather evaluate PREON. The finalists can be split into two rough categories: Blake, Keccak and Skein are AXR-based, while Grøstl and JH are AES-based. For the cases of Blake and Keccak, Hoerder et al. [15] highlighted the difficulty of finding appropriate ISEs: their design is already RISC-friendly and potential ISEs therefore fit a more coprocessor-like approach that captures and operates on (most of) the state in each step.

JH-256 We pack eight 4-bit state words into a 32-bit register and utilize three ISEs: one for the S-box and linear transform layer, and two more for the permutation layer; the design of JH means the same ISEs can be used for all parametrisations. To maintain comparability with the reference implementation, we use the ISEs to compute the round constants at run-time. This requires re-packing the round constants at various points, and explains the marginal increase in code footprint.

Skein-512-256 We focus on acceleration of the internal Threefish cipher. In order to match the 32-bit datapath of the LEON3, we specify a four-step ISE to support the MIX function; for comparison, we use the 32-bit oriented reference implementation. To avoid the need for a general-purpose rotation unit, we specify (and supply immediate inputs to select) ISEs for each of 27 rotation distances. The disadvantage of this approach is that the same ISEs can not support a different parametrisation.

Grøstl-256 We pack four 8-bit state words into a 32-bit register and use three ISEs: one for the `SubBytes` step, and two more for the `MixBytes` step. The T-tables based reference implementation is used for comparison.

Multiplication in $\mathbb{Z}_N^*$ (supporting RSA, ECC). Großschädl et al. [11] propose an ISE for RSA, or more specifically for Montgomery multiplication; their design is implemented on a SPARC V8-based LEON2. The ISE focuses on Multiply-ACcumulate (MAC) operations (e.g., $S \leftarrow S + a \times b$), and uses three dedicated 32-bit accumulator registers. We replicate this approach fairly directly, housing the accumulators within the fabric configuration itself, and compare our results with a C implementation provided by the authors of [11].

Multiplication in $\mathbb{F}_{2^n}$ and $\mathbb{F}_{3^m}$ (supporting ECC, Pairing-Based Crypto). Considering the cases $n = 233$ and $m = 337$, arithmetic in $\mathbb{F}_{2^{233}}[X]/X^{233} + X^{73} + 1$ and $\mathbb{F}_{3^{337}}[Y]/Y^{337} + Y^{30} - 1$ underpin specific parametrisations in elliptic curve and pairing-based cryptography, e.g. the former is specified in NIST-B-233. In the characteristic-two case, coefficients of some $x \in \mathbb{F}_{2^n}$ have a natural representation; various published ISEs, including those for SPARC V8-based LEON2 [30] and the Intel CLMUL extension to x86, provide an associated polynomial (or "carry-less") multiplication instruction. A similar concept is possible in the characteristic-three case, but the issue of representation is more complex.

Our implementations accelerate school-book multiplication mainly via a dedicated polynomial multiplication ISE; the width-4 comb based multiplication is accelerated using a "shift with carry" ISE, which is missing in the SPARC V8 instruction set. Particularly for characteristic three, the flexibility of PREON is beneficial: it allows a range of subtle implementation options without changes to the architecture, and resolves some problems with previous work. For example Grabher et al. [10] support bit-slicing, but only using unconventional 6-address instructions; the "operand fetch" idiom in PREON can cope with multi-operand instructions, and still provide significant performance improvements.

Summary and Discussion. It is unsurprising that ISE-based implementations improve either latency (overall cycles) and/or size (memory footprint). For the kernels studied, the latter case implies a hidden side-benefit of reducing memory traffic (primarily loads) and hence reduced reliance on a cache to achieve quoted performance; in a rough sense, one can view the cost of including the fabric as counterbalancing the need for large, efficient layers of memory hierarchy.

Focusing on AES, one can contrast results for dynamic ISE provided by the PREON re-configurable fabric with static ISE (taking AES-NI as an example) and coarsely-integrated FPGA-based coprocessors. As stated in [34], AES-NI achieves a throughput of 3.589 cycles per byte (in CBC mode), i.e., 57.4 cycles per block. However, AES-NI operates on 128-bit SSE registers, which increases the throughput by a factor of 4 in relation to a 32-bit bit datapath. Hence, one can estimate that a 32-bit analogue of AES-NI would require about 230 cycles per block, roughly half that of the PREON-supported implementation. Comparison with an FPGA-based cryptographic coprocessor must consider the interface through which coprocessor and host are connected. For example, Hodjat et al [14] and Schaumont et al. [27] attached an AES coprocessor to the LEON core and achieved a throughput of 704 cycles (via the LEON coprocessor interface) and 1492 cycles (via memory-mapped I/O) per block, even though the AES core itself performs an encryption in only 11 cycles. Using a dedicated high-speed interconnect, such as Xilinx's Fast Simplex Link (FSL), allows improvement to about 202 cycles per block [9]. In summary, even accepting the limited scope and accuracy of this comparison, the PREON prototype offers a very attractive compromise: it is competitive to both static AES extensions *and* coprocessors using metrics of performance and flexibility.

This flexibility is highlighted by the possibility for combination of configurations to support multi-kernel ISEs. Trivial merging of configurations is possible where size permits, but more specific approaches also exist. For example, one may consider multi-field multipliers and, hence, multi-kernel ISEs as described by Vejda et al. [33]. As a more concrete example, AES and Grøstl compute the S-box function in the same way, and therefore it is possible to design a single configuration that supports both kernels. While the performance figures remain the same as above, the total size required is 152 slices, roughly 30% less than a separate implementation.

A less positive issue is that of re-configuration speed. Ideally, one might aim for per-instruction change in an ISE (i.e., the configuration) to maximise the emphasis on flexibility. However, without the facility for multiple resident configurations, the re-configuration speed of our Xilinx Virtex-5 FPGA (100 Mbit/s at 25 MHz, meaning latencies of upto 1000 µs for our more complex cases) is a limiting factor. In a sense this is a property of the technology used to realise the fabric, but even so the re-configuration speed limits "ISE dynamism," which one might reasonably argue is disadvantageous.

4.2 Instruction Register File

Hines et al. [13, Table 2] include a set of security-related benchmarks (e.g., AES and SHA-2), adopting a domain-neutral (and compiler supported) approach to implementation. Since our design differs, we do not offer a direct comparison with these results. Rather, we aim to explore how careful use of our IRF design compares with the natural alternative of cached instruction access.

Again, we use the sub-sections below to discuss each implementation, and conclude with a summary of the results.

Table 2. Experimental results comparing the performance of various cryptographic workloads without and with support from ISEs provided by the IRF (and without and with support from the I-cache in each case). Note that initialisation of the IRF buffers are not included in the total number of cycles.

AES				
	Without ISE		With ISE	
	Performance (cycles)	Fetches (from main memory)	Performance (cycles)	Fetches (from main memory)
Without I-cache	4751	823	2644	309
With I-cache	1281	823	1302	309
Multiplication in $\mathbb{Z}_N^*$				
	Without ISE		With ISE	
	Performance (cycles)	Fetches (from main memory)	Performance (cycles)	Fetches (from main memory)
Without I-cache	189069	34765	67366	4046
With I-cache	51708	34765	48640	4046

AES. Parametrising the IRF with $m = 4$, $n = 16$, we record each “block” of a T-tables based AES implementation into a buffer; these are then replayed (as roughly illustrated in Figure 1) to form each round.

Multiplication in $\mathbb{Z}_N^*$ (supporting RSA, ECC). Parametrising the IRF with $m = 4, n = 16$, we refer directly to the CIOS algorithm in [4, Section 5]. The idea is to record the body of each inner loop into a buffer; we include a final instruction which increments j. Then, by using one `playback` instruction, the buffer can be replayed s times (having set $C = 0$ and $j = 0$ initially); the resulting, expanded instruction sequence implements the entire unrolled loop.

Summary and Discussion. Table 2 outlines our results. Broadly speaking, the conclusion is that use of the IRF can significantly reduce the number of fetches from memory (roughly 2- and 8-fold improvement) without significant negative impact (indeed, in some cases with positive impact) on the performance. The latter result is, naturally, magnified when we switch off the I-cache and (e.g., in the case of Montgomery multiplication) realise benefits of loop unrolling without the associated disadvantage in terms of static footprint.

In an attempt to quantify this in terms of power consumption, we refer to the widely cited figures provided by Segars [28, Slides 34 and 42]. He quotes an ARM9TDMI register file as representing 13% of the datapath power consumption, and ARM920T I- and D-caches as 25% and 19%, respectively, of the total power consumption. Therefore, one can estimate that an ARM920T register file represents about 13% of the quoted 25% total power consumption. As such, one might roughly reason that replacing the I-cache with an IRF reduces the total power consumption by around 20% (per fetch from IRF-resident content).

4.3 Combined Utilisation

A key motivations for our specific selection of mechanisms above is the potential for composing their use: since use of the fabric is via a normal instruction, such instructions can be captured in the IRF like any other. As a final, one-off case study, we produced such a combined implementation of AES. Encryption of a 128-bit block using the T-tables based reference implementation performs 823 instruction fetches from main memory, 208 loads and 4 stores; it takes a total of 1281 cycles and relies on a 6068 B static memory footprint. In contrast, an ISE-based implementation (combining the previous fabric configuration *and* the IRF parametrised with $m = 1, n = 64$) performs 45 instruction fetches from main memory, 48 loads and 4 stores; it takes a total of 434 cycles and relies on a 340 B static memory footprint.

In summary, considered use of the two mechanisms yields a 3-fold improvement in performance, a 10-fold improvement in main memory access (combined fetches, loads and stores), and a 17-fold improvement in memory footprint; this is achieved in a manner which permits similar benefit to other kernels without alteration of the PREON architecture, and largely without the I- and D-caches which underpin the T-tables based approach.

5 Issues Relating to Practical Deployment

Section 4 highlights some practical advantages of the two mechanisms considered. However, these advantages rely on the re-configurable fabric and IRF in PREON housing state: their configuration in both cases, and internal registers in the case of the fabric. As such, one must *also* consider related disadvantages (i.e., the issue of security). In this section, we discuss some (fairly speculative) examples within the context of both PREON and other existing proposals.

Trusted Configuration and Use. Design verification and policy enforcement are well-researched areas in embedded security, and form a key requirement within high-assurance contexts. A review of related techniques is, for example, given by Huffmire et al. [16, Sect. 4.4.1]; they point out that partial re-configuration is rarely used within said contexts due to the increased complexity of design verification. However, the approach of coupling a general-purpose processor to a re-configurable fabric offers a potential solution to this dilemma. Since access control to the re-configurable fabric *must* be integrated into the security model of the processor, this will not cause the same problem as the more general case considered by Huffmire et al. In particular, it is no more difficult to design access policies for the fabric than for the processor itself.

As an example, access to the re-configurable fabric might be limited to the OS kernel via a privilege mode within the processor; this offers a similar protection to that for conventional process state. Likewise, the processor may enforce policies on the configuration bit-stream; a possible approach is to accept only authenticated bit-streams. An effective implementation of such mechanisms is fundamental to mitigation of several problems outlined below.

State "Read Out". Focusing on the fabric, it is obvious that internal registers maintained by one process should not be readable by any other process. More subtle issues are raised by the fabric configuration itself. For example, pushed to an extreme, it is tempting to consider aggressive compile-time or run-time specialisation techniques (cf. Warp processors [21]), e.g., a fabric configuration specialised per key. In this case, it is also vital that the configuration can not be read by another process (or by an external attacker): Kerckhoffs' principle does not apply if secret key material is embedded in the configuration rather than simply used by it. This issue relates vaguely to attacks described in [39].

Information Leakage. When unmanaged, the state of the re-configurable fabric acts as a shared resource between processes. Said resource (or conversely, the lack of appropriate process isolation) represents the potential for various forms of micro-architectural attack; for example, see Wang and Lee [35, Section 4].

A concrete example can be applied to the ProteanARM [6], which requires a process to register the fabric configurations with the OS. When an instruction references a configuration (via an identifier), it is either

1. executed by the fabric (if the configuration is resident),
2. transformed into a call to an equivalent software implementation (where the registration dictates this mode), or
3. causes an exception (whereby the OS can load the configuration if it is not resident).

In a rough sense, this implies that the content of the re-configurable fabric can be "queried" by timing how long a use of the fabric takes to complete. Hence, a heavy-weight version of the so-called "prime + probe" approach to cache-based side-channel attacks seems to apply.

Of course, it is possible to construct mitigating solutions. For example, one can (at least in theory) demand a full context switch of all such resources. In practise, however, it is often very tempting to take short-cuts since the overall cost of switching the context of the re-configurable fabric is extremely high (as illustrated by Section 4). We note that Chan et al. [5] examine a similar issue of processor isolation albeit in a more coprocessor-like context.

Fault Injection. Although one can question whether the statement is still true today, in 2003 Wollinger and Paar [36] mention that "there appears to be no published attempt to perform this kind of [fault] attack against FPGAs." A cursory literature search shows there is (at least) not as much work in this area as one might expect, a notable exception being [3], with more attention paid to ASICs. Some related work includes that of Desmedt et al. [7] and Hadžic et al [12], who introduce the idea of an FPGA "virus." If one accepts mechanisms to support some type of re-configurable fabrics as a viable direction, the examples above suggest issues in terms of security. In particular, does the re-configurable nature of an FPGA mean that "traditional" fault attacks on computation are easier (e.g., by altering logic cells in the same way as RAM)?

Hardware Trojans. In order to reduce the cost of a context switch, the ProteanARM processor [6, Sect. 4] allows a configuration associated with one process to be resident in the fabric while another process is being executed. Imagine the re-configurable fabric is not gated, i.e., that operands are fed to it and computation occurs even if the output is not used. Since the fabric is not forcibly re-configured for each process, a process may unintentionally provoke computation within the fabric whose configuration is dictated by another process. Or, imagine two (partial) configurations coexisting on a fabric: one might speculate that the behaviour of one (e.g., with respect to thermal properties) could influence the other in some way. These simple examples suggest the potential for a hardware Trojan: the attacker configures the fabric with a high-leakage function which is able to capture (or export) information leaked by some target process.

As above, the issue of isolation is important. We note that the "moats and drawbridges" design concept of Kastner et al. [18] is of special interest in this context: the goal is physical in-fabric isolation of partial configurations, i.e., the separation of Trojan hardware from benign targets.

6 Conclusions

Our results in Section 4 highlight advantages with respect to support for and use of dynamic ISEs in cryptography. Both conceptually simple, relatively non-invasive additions to the LEON3 generalise many existing ISE proposals for this platform and permit high-performance, "algorithm-agile" implementations. In short, such an approach can support ISEs like AES-NI without a need for fixed AES-related functionality. However, to realise said advantages, and following a similar line of reasoning as [20, 26], we show in Section 5 that careful analysis and consideration of security is a strict prerequisite.

At least two well-founded criticisms exist. First, one might view speculative attacks against prototype processor designs as moot. Second, and focusing on the re-configurable fabric, one may argue that other design constraints prevent integration of an FPGA into the processor data-path. Recalling Section 2, we again stress that processors of this sort *already* exist; in a sense, commercialised examples offer an interesting vehicle for future work on some questions raised in Section 5. Along similar lines, we stress that it is perfectly viable to instead find a compromise between general- and special-purpose fabric: again, this is an interesting challenge for future work. Partly re-configurable functional units (e.g., those in PipeRench [29] and CryptoManiac [37]) give some direction.

Even though there are some unresolved challenges, our experimental results suggest that the general concept of providing tightly integrated re-configurable components represents an interesting approach for (embedded) processors. We further conclude that provision of exposed, programmer-controlled components rather than automated (cf. the transparent operation of caches) alternatives is an attractive direction for cryptography since they allow at least the potential to avoid classes of existing micro-architectural (e.g., cache-based) attack.

Acknowledgements. The work described in this paper has been supported in part by EPSRC grant EP/H001689/1. The authors would like to thank Atukem Nabina for his general input on FPGA partial re-configuration.

References

1. H. Amano. A survey on dynamically reconfigurable processors. *IEICE Tran. Comm.*, E89-B(12):3179–3187, 2006.
2. R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *CODES*, pages 73–78, 2002.
3. G. Canivet, P. Maistri, R. Leveugle, J. Clédière, F. Valette, and M. Renaudin. Glitch and laser fault attacks onto a secure AES implementation on a SRAM-based FPGA. *J. Cryptology*, 24(2):247–268, 2011.
4. Ç.K. Koç, T. Acar, and B.S. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, 1996.
5. H. Chan, P. Schaumont, and I. Verbauwhede. Process isolation for reconfigurable hardware. In *ERSA*, pages 164–170, 2006.
6. M.W. Dales. *Managing a reconfigurable processor in a general purpose workstation environment*. PhD thesis, University of Glasgow, 2003.
7. Y. Desmedt and J.-J. Quisquater. Public-key systems based on the difficulty of tampering (is there a difference between DES and RSA?). In *CRYPTO*, pages 111–117. LNCS 263, 1986.
8. M.J. Flynn and M.D. McLaren. Microprogramming revisited. In *Proc. of the 22nd ACM National Conference*, pages 457–464, 1967.
9. I. Gonzalez and F. Gómez-Arribas. Ciphering algorithms in MicroBlaze-based embedded systems. *Computers and Digital Techniques*, 153(2):87–92, 2006.
10. P. Grabher, J. Großschädl, and D. Page. Light-weight instruction set extensions for bit-sliced cryptography. In *CHES*, pages 331–345. LNCS 5154, 2008.
11. J. Großschädl, S. Tillich, and A. Szekely. Performance evaluation of instruction set extensions for long integer modular arithmetic on a SPARC V8 processor. In *DSD*, pages 680–689, 2007.
12. I. Hadžic, S. Udani, and J.M. Smith. FPGA viruses. In *FPL*, pages 291–300. LNCS 1673, 1999.
13. S.R. Hines, J. Green, G. Tyson, and D. Whalley. Improving program efficiency by packing instructions into registers. In *ISCA*, pages 260–271, 2005.
14. A. Hodjat and I. Verbauwhede. Interfacing a high speed crypto accelerator to an embedded CPU. In *Asilomar Conference on Signals, Systems, and Computers*, volume 1, pages 488–492, 2004.
15. S. Hoerder, M. Wójcik, S. Tillich, and D. Page. An evaluation of hash functions on a power analysis resistant processor architecture. In *WISTP*, pages 160–174. LNCS 6633, 2011.
16. T. Huffmire, C. Irvine, T.D. Nguyen, T. Levin, R. Kastner, and T. Sherwood. *Handbook of FPGA Design Security*. Springer, 2010.
17. M. Juliato and C. Gebotys. Tailoring a reconfigurable platform to SHA-256 and HMAC through custom instructions and peripherals. In *ReConFig*, pages 195–200, 2009.
18. R. Kastner, T. Levin, T. Nguyen, C. Irvine, B. Brotherton, G. Wang, T. Sherwood, and T. Huffmire. Moats and drawbridges: An isolation primitive for reconfigurable hardware based systems. In *IEEE Security and Privacy*, pages 281–295, 2007.
19. T. Kluter, P. Brisk, P. Ienne, and E. Charbon. Way stealing: cache-assisted automatic instruction set extensions. In *DAC*, pages 31–36, 2009.
20. P.C. Kocher, R.B. Lee, G. McGraw, and A. Raghunathan. Security as a new dimension in embedded system design. In *DAC*, pages 753–760, 2004.

21. R. Lysecky, G. Stitt, and F. Vahid. Warp processors. *TODAES*, 11(3):659–681, 2006.
22. N. Malik, R.J. Eickemeyer, and S. Vassiliadis. Interlock collapsing ALU for increased instruction-level parallelism. *SIGMICRO Newsletter*, 23(1-2):149–157, 1992.
23. J.E. Miller and A. Agarwal. Software-based instruction caching for embedded processors. In *ASPLOS*, pages 293–302, 2006.
24. C.R. Moore, D.M. Balser, J.S. Muhich, and R.E. East. IBM single chip RISC processor (RSC). In *ICCD*, pages 200–204, 1991.
25. N. Pothineni, P. Brisk, P. Ienne, A. Kumar, and K. Paul. A high-level synthesis flow for custom instruction set extensions for application-specific processors. In *ASP-DAC*, pages 707–712, 2010.
26. S. Ravi, A. Raghunathan, P.C. Kocher, and S. Hattangady. Security in embedded systems: Design challenges. *TECS*, 3(3):461–491, 2004.
27. P. Schaumont, K. Sakiyama, A. Hodjat, and I. Verbauwhede. Embedded software integration for coarse-grain reconfigurable systems. In *IPDPS*, pages 137–142, 2004.
28. S. Segars. Low power design techniques for microprocessors (tutorial session). In *ISSCC*, 2001.
29. R.R. Taylor and S.C. Goldstein. A high-performance flexible architecture for cryptography. In *CHES*, pages 231–245. LNCS 1717, 1999.
30. S. Tillich and J. Großschädl. A simple architectural enhancement for fast and flexible elliptic curve cryptography over binary finite fields $GF(2^m)$. In *ACSAC*, pages 282–295. LNCS 3189, 2003.
31. S. Tillich and J. Großschädl. Instruction set extensions for efficient AES implementation on 32-bit processors. In *CHES*, pages 270–284. LNCS 4249, 2006.
32. A.B. Tucker and M.J. Flynn. Dynamic microprogramming: processor organization and programming. *CACM*, 14(4):240–250, 1971.
33. T. Vejda, D. Page, and J. Großschädl. Instruction set extensions for pairing-based cryptography. In *Pairing-Based Cryptography*, pages 208–224. LNCS 4575, 2007.
34. VeriSign. An evaluation of new processor instructions for accelerating selected cryptographic algorithms, 2010.
35. Z. Wang and R.B. Lee. Covert and side channels due to processor architecture. In *ACSAC*, pages 473–482, 2006.
36. T. Wollinger and C. Paar. How secure are FPGAs in cryptographic applications? In *FPL*, pages 91–100. LNCS 2778, 2003.
37. L. Wu, C. Weaver, and T. Austin. CryptoManiac: a fast flexible architecture for secure communication. In *ISCA*, pages 110–119, 2001.
38. Xilinx. Partial reconfiguration user guide (UG702) v12.1, 2010. `http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_1/ug702.pdf`.
39. B. Yang, K. Wu, and R. Karri. Scan based side channel attack on dedicated hardware implementations of data encryption standard. In *ITC*, pages 339–344, 2004.