SME: Scalable Masking Extensions

Toward software portability and hardware security for masking countermeasures.

Ben Marshall[1,2] and Dan Page[1]

[1] Department of Computer Science, University of Bristol,
Merchant Venturers Building, Woodland Road,
Bristol, BS8 1UB, United Kingdom.
{ben.marshall,daniel.page}@bristol.ac.uk
[2] PQShield Ltd, Oxford
ben.marshall@pqshield.com

Abstract. Supporting masking countermeasures for non-invasive side-channel security in instructions set architectures is a hard problem. Masked operations often have a large number of inputs and outputs, and enabling portable higher order masking has remained a difficult. However, there are clear benefits to enabling this in terms of performance, code density and security guarantees. We present SME, an instruction set extension for enabling secure and efficient software masking of cryptographic code at higher security orders. Our design improves on past work by enabling the same software to run at higher masking orders, depending on the level of security the CPU/SoC implementer has deemed appropriate for their product or device at design time. Our approach relies on similarities between implementations of higher order masking schemes and traditional vector programming. It greatly simplifies the task of writing masked software, and restores the basic promise of ISAs: that the same software will run correctly and securely on any correctly implemented CPU with the necessary security guarantees. We describe our concept as a custom extension to the RISC-V ISA, and its soon to be ratified scalar cryptography extension. An example implementation is also described, with performance and area tradeoffs detailed for several masking security orders. To our knowledge, ours is the first example of enabling flexible side-channel secure implementations of the official RISC-V lightweight cryptography instructions.

Keywords: side-channel attack, micro-architectural leakage, instruction set extension, masking, RISC-V

1 Introduction

Writing power side-channel secure software is hard. While tooling to aid developers in this has progressed significantly recently [BBYS21], there remain several challenges to making these tools *portable* across the multitude of different devices used to create side-channel secure products in industry, or which are used as the basis of research in academia.

A significant problem with the portability of these tools lies in how different devices have very different leakage characteristics, even though they may implement the same Instruction Set Architecture (ISA). These micro-architectural differences are well documented [MPW21, SSB+19], and undermine masked implementations by causing un-expected share collisions. This forces any software which tries to be portably secure to be extremely conservative in it's assumptions about underlying devices, often making it unacceptably large (in terms of code size) or slow.

Similarly, increasing the masking order of an implementation (perhaps to increase the security margin due to micro-architectural power leakages) often leads to a substantially larger executable code size, which is too large for an embedded device for which cryptography is only a small part of its actual function. Further, there has been no way for existing software to automatically and portably take advantage of improvements in security and performance guarantees offered by hardware in the context of masking countermeasures.

One way to view this is as a problem stemming from the ISA abstraction. This abstraction has proved to be exceptionally good at guaranteeing *functional* characteristics across devices, but was never designed to give guarantees about analog side-channel security. Recent works have sought to repair or augment the ISA by adding instructions which give software control over leaking micro-architectural state [GMPP20], by adding dedicated instructions which explicitly require some level of side-channel security in their implementation [KS20, GGM+20], or by creating a masked representation of the entire data-path transparently to software [GJM+17, DMGH19]. These approaches have achieved some successes, but suffer from two common problems: extending to higher masking orders, and avoiding the code size and performance penalty associated with higher order masking.

In this work, we note some similarities between operating on shared representations of variables, and the kind of scalable vector programming paradigms which first appeared many years ago, and are currently being popularised by SVE [SBB+17] from ARM and the RISC-V Vector Extension [ris]. Stemming from this observation, the main contributions of this work are:

- A concrete *architectural* design for allowing support for scalable masking orders in a CPU, such that the same software can execute on any CPU at some maximum supported masking order, while retaining all of the associated security guarantees. This includes techniques for automatically managing the additional state required by masked software.
- A discussion of design issues which an ISE must consider when designing for side-channel security, and how the architecture can promote or encourage side-channel secure software design.
- An example implementation of this architecture, based on RISC-V and the open source SCARV-CPU core.
- A performance evaluation of the implementation at multiple masking orders for important cryptographic workloads, including (to our knowledge) the first side-channel secure implementation in hardware of the official RISC-V scalar cryptography extensions for AES [MNP+20].

We focus without loss of generality on power side-channel leakage, and emphasise that we consider the *architecture* as the main contribution. Our main aim is to define an appropriate *interface* which can be easily implemented in hardware, and which provides all of the security guarantees necessary for software as part of its definition. Hence, it is the interface definition which is more important than any one implementation, though we nonetheless show empirically that our interface can be implemented realistically. Our hope is that the concrete architecture definition allows those with expertise above (in terms of formal modelling) and below (in terms of masked hardware implementations) the ISA to experiment with, extend and improve our design.

2 Background

RISC-V. RISC-V is a (relatively) new instruction set architecture designed around a very small core set of simple instructions, which can be extended with domain specific extensions. For example, the `F` extension adds Floating Point support, or the `C` extension adds 16-bit instructions for improved code density. The base ISA comes in three main variants: 32, 64 or 128-bit, denoted by XLEN. User-mode applications can access 31 general

purpose registers, and a *zero* constant register. Privileged applications can also access the Control and Status Registers (CSRs), which control things like interrupts, virtual memory and exception handling. A small number of CSRs are also visible to user-mode applications for managing things like floating point flags.

Also of note is the soon to be ratified RISC-V Scalar Cryptography extension [ZGS+21]. This adds support for hardware accelerated cryptography using only the general purpose register file, *without* needing large SIMD or vector registers. The combination of being free to implement, and the first ISA to add lightweight hardware accelerated cryptography means that RISC-V is likely to see wide adoption in embedded platforms with side-channel security requirements.

Vector v.s. SIMD programming. These are two overlapping but distinct notions in computer architecture. In SIMD programming, one typically uses a single instruction to operate on multiple pairwise elements of a wide register. Often, SIMD instructions have been added to architectures to opportunistically support parallel operation on narrower data types than the machine word width. E.g. DSP cores enabling two 16-bit elements stored in a single 32-bit register to be added together in one instruction. This is sometimes called "SIMD within a register" or *SWAR*. The key characteristic of SIMD programming is that there is little support for *arrays* of elements which are not an integer multiple of the number of elements which can be operated on by a single SIMD instruction. This leads to a considerable amount of "loop maintenence" code.

Vector programming is similar to SIMD in that it allows a single instruction to operate on multiple data items. However, vector processors include explicit support for managing arbitrary length vectors. This is done by making the maximum number of elements which can be processed in hardware available dynamically to software, which considerably reduces the amount of loop management code which is common with SIMD instructions.

2.1 Related Work

Here we describe some of the closest works to our own and how they compare in terms of scope and aims.

In [GJM+17], the authors describe an augmented RISC-V CPU which can also protect software from power side channel attacks using masking. It does this by separating the data-paths, and adding dedicated hardware support for certain masked operations. A DOM [GMK16] representation is used to implement the core masked operations, which are versions of the the basic RISC-V ALU instructions. Their design also supports a synthesis-time parameter to change the masking order, and the paper includes overhead results using upto four shares. The entire register file uses a shared representation. Though not explicitly described in the paper, we believe individual shares are loaded or stored as needed, with some un-described mechanism to select which share is stored or loaded. This is also no described way for software to determine how many shares are used to represent variables, meaning software is not portable between instances of a design with different numbers of hardware supported shares. We would hence classify this work as being "below" the ISA boundary. That is, software cannot interrogate the hardware support for masking, requiring a human to write code specific to it.

In [DMGH19] the authors describe a commercial solution for an entirely masked CPU implementation, where the "register file, the data-path including the ALU, memory, and the memory interface" are all protected using a masked representation. Software is unaware of the masking, allowing native software to work transparently, which is a serious advantage in terms of programmer effort. However, only 1'st order masking is supported. An interesting solution to transparently protecting shared variables in memory is described. An internal "seed" value is used to generate an extra share based on the seed and the memory address. This share is used to collapse the internal 2-share representation into a single variable,

which can be turned back into a 2-share representation on a load by re-generating the additional share from the seed and address again. This removes the usual memory overhead associated with masking, where additional shares increase the memory requirement with the masking order. There are no concrete hardware overhead numbers for this design, we assume because they are commercially sensitive.

Both [KS20] and [GGM+20] describe complete instruction set extensions for supporting $1'st$ order masking within the CPU. [GGM+20] includes a complete engineering prototype, and a side-channel evaluation of several algorithms including AES. While these examples show well how 1'st order masking can be supported, both point out that supporting higher order masking is an open question. One particular problem in [GGM+20] is the difficulty of safely storing multiple shares in the general purpose register file without causing accidental unmasking. Solving this required considerable engineering effort in the micro-architecture, stemming from the fact that *architecturally* combining shares was valid behaviour for the architecture.

3 Architecture

Here, we describe our design as an extension to the base RISC-V ISA. The key concept to SME is that any CPU implementing it provides support for upto some maximum *fixed* number of shares, which software can determine at runtime. So long as the software is written according to some basic rules, it can be executed on any CPU implementing SME, with upto d-order security, where d is the maximum number of shares supported in hardware by the CPU with no changes to the software itself.

This is *similar* to the standard notion of vector programming, as adopted by the RISC-V vector extension [ris]. While our extension could possibly be built *on top of* the base RISC-V vector extension, we opted for a dedicated, orthogonal design for the purposes of explaining it. This was to avoid much of the complexity and additional state which the RISC-V vector extension adds, though we hope similarities are clear to the reader, and leave open the possibility of a merged design in the future. A more detailed discussion on the similarities and differences between SME and the RISC-V vector extension is found in Section 6.

While designing the SME extension, we tried to follow these principles:

1. Add as little new architectural state as possible to achieve useful 1'st order security. Any more architectural state added after that must directly contribute to increasing the maximum security order.
2. Keep the programmer model as simple as possible. The programmers life should not be made more complex with an increasing security order.
3. New instructions should read at most two general purpose registers and write at most one. This is in keeping with the existing RISC-V principles. Likewise, any additional state should not require register files with more than two architectural read ports and one architectural write port.
4. Allow for as much implementation diversity as is useful, with tradeoffs between area, performance and code size explicitly noted.
5. Avoid the temptation to *bolt on* functionality which would obviously be better implemented as a memory mapped co-processor[1].
6. Design something which could be commercially viable or standards appropriate. In terms of building a practical implementation, this means giving SoC builders a reasonable tradeoff between size, performance, security and efficiency. A good guide is to take the

[1] While this approach can give dramatic performance improvements for little area cost by (e.g.) reusing the general purpose registers files, it can be a disaster for wider system efficiency when *not* doing a cryptographic operation.

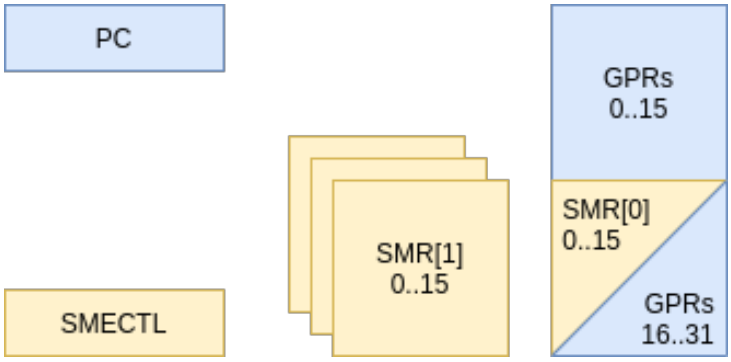

Figure 1: Diagram of base RISC-V architecture state, and additional state due to SME.

area required for dedicated side-channel secure ECC, AES and SHA accelerators, and re-purpose a comparable amount of silicon area for SME. See Section 3.5 for a short discussion on this.

7. Consider interactions with other parts of the RISC-V ISA, re-using functionality where appropriate and avoiding additional complexity.

All implementations of SME define an implementation specific constant SMAX. This is the maximum number of physical shares supported by the SME implementation. We *deliberately* do not call this a *security* order. This is because different masking schemes require different numbers of shares to give the same security, and SME*does not* mandate a particular masking scheme.

It is essential to de-couple the exact masking scheme from the architecture and programmer model, as it allows for future schemes which are more efficient or secure to be implemented without architectural changes to SME. Or, it enables a degree of agility if an existing scheme is found to be insecure.

3.1 SME State

A diagram of additional state added by SME is shown in Figure 1. To control the SME extension, we introduce a single user-mode Control and Status Register (CSR) to the RISC-V architectural state called smectl, shown in Figure 2. The fields in smectl are:

1. smectl.D - *d-share security*. This is a read/write field which indicates the number of shares to use. A value of 0 indicates SME is currently turned off. A value of 2 indicates a two share scheme, 3 indicates a three share scheme etc. A value of 1 is currently *reserved*. Only values upto SMAX may be written to smectl.D. Note that implementations may make smectl.D support only a single writable value, in which case the number of shares for any computation will always be fixed.
2. smectl.T - *Masking Type*. When zero, this bit indicates that *boolean* masking is being used. When set, it indicates *arithmetic* masking is being used. This bit controls the behaviour of the masking instructions described in Section 3.3.2, and modifies some existing instructions in Section 3.2.
3. smectl.B - *Share Bank*. This read/write field may hold values between 0 and smectl.D. It is used by load and store instructions (described later) to select which set of shares to load or store to memory. Writing smectl.B with an invalid value raises an Invalid Opcode Exception.
4. smectl.p* - *Previous Fields*. These fields are used to gracefully handle traps and interrupts. Their operation is described fully in Section 3.1.1.

Using smectl., software can configure the number of shares to use to fit some perfor-

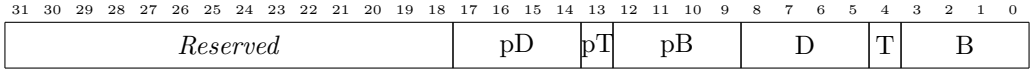

Figure 2: Layout of the register for XLEN =32. For XLEN =64, bits 63 : 32 are *reserved*.

mance/security trade off. Or, software can write −1 (i.e. all ones) to smectl.D, and always use the maximum number of shares available. For our initial description of SME, the maximum supported SMAX is 16. A well defined *off* state for SME means that implementations can (for example) clock gate registers efficiently when they are not in use, or even power down that part of the chip completely.

Note that the low five bits of the smectl register can be set/cleared with a single RISC-V `csr*i` instruction.

An implementation of SME will also include SMAX *banks* of registers used to hold shares. Each bank contains 16 registers. Each share register is denoted $\mathsf{SMR}_X[Y]$, where X is the *bank number* and Y is the Y'th register of the bank. Bank 0 is an alias for the top 16 base RISC-V GPRs, while Bank $N > 1$ is additional state which is only accessible by SME aware software. Hence, a variable X, with its 0'th share in $\mathsf{GPR}[i]$ is represented as smectl.D shares: $X = \{\mathsf{SMR}_0[i], \mathsf{SMR}_1[i], ..., \mathsf{SMR}_D[i]\}$.

We deliberately chose 16 registers per bank, rather than to simply have SMAX coppies of the GPR register file. This was to recognise that when implementing a masked cryptographic primitive, not all registers are ever used to hold shares: some are used for loop counters and array addresses. Hence, we use the top 16 RISC-V GPRs to represent the 0'th share of any variable, and additional register banks for i'th additional shares. GPRs 0..15 can then be used to hold loop counters, pointers, return addresses and so on[2].

Further, registers and flip-flops often dominate the area of small CPU implementations and side-channel secure hardware operations. Duplicating all 31 GPRs would give 1 additional share, with less than optimal register utilisation. After all, who cares about shared representations of the stack pointer? Our scheme allows for 3 share variable representations with the same number of additional architectural registers, and better register utilisation for sensitive variables.

We believe being able to hold 16 sensitive variables inside the CPU core at any one time is enough, given that the ARMv7-M architecture so often used in side-channel secure implementation research has only 12 general purpose registers in total. Our empirical evaluations in Section 5 suggest that while there are corner cases which result in additional register spilling, even the worse case is considerably better than the status quo for software masked implementations.

3.1.1 Interrupt and Trap Handling

Any addition of state to an ISA inevitably leads to the cry of "*what about interrupt latency?*" or "*what about context switches?*". We have two responses in this case. The first, is that interrupts *may* be disabled for the duration of a cryptographic operation, though this is hardly ideal and not always possible. Second, we provide an efficient (three instructions) way to check if SME is in use by the current thread, so SME related state only needs to be saved/restored as needed. This is similar to the mechanism RISC-V provides for checking if floating point state needs to be saved or restored.

Further, we add dedicated support for trapping through the smectl.p* fields of smectl.. Whenever a trap or interrupt occurs, the following happens inside smectl.: All smectl.p* fields take on the current value of the corresponding smectl.D/T/B field. The smectl.D field is then set to zero, turning off SME for the duration of the trap or interrupt handler. Software may interrogate the smectl.p* fields to discover the previous operating state

[2]One could choose a more complex sharing of the RISC-V GPRs and SME registers, optimising for the RISC-V ABI and register usage conventions. We leave this as a clear optimisation for the future.

of SME, and save registers as needed. This mirrors how the base RISC-V Privileged Architecture handles interrupt enable and privilege level signals across traps and interrupts. If the servicing routine itself does not need to use SME, then it may safely save and restore any registers of SME Bank 0 (i.e. those that overlap with the GPRs) without worrying about share collisions, as other shares are safely locked away in other Banks.

Upon returning from a trap or interrupt service routine, the values of smectl.D/T/B have their values restored from the corresponding smectl.p* fields, and execution may continue as normal.

We belive graceful interrupt and trap handling for instruction set extensions which add architectural state is often an underdeveloped point in many works. For example, [GGM+20] contains no discussion on the topic, and we believe the design would trivially leak during an interrupt service routine which sequentially stored GPRs to the stack. We hope the considerations here offer routes for handling this in other works.

3.2 Modifications to Existing Instructions

Rather than adding new *masked* variants of base ISA instructions, as was done in [GGM+20], instead we overload the behaviour of existing base ISA ALU instructions. We do this, because it re-inforces the notion of SME adding a side-channel secure *mode* to the host CPU core. It also preserves opcode space, which is an essential consideration for standards appropriate extensions. In reality however, it would be perfectly reasonable to realise SME using new, dedicated instruction encodings if the wider system constraints make sense. For the overloaded instructions, when smectl.D is non-zero:

- ALU instructions which write to and source only the top 16 RISC-V GPRs (I.e. $\mathsf{SMR}_0[i]$) automatically perform a smectl.D-share version of their operation.
- If an ALU instruction performs a linear operation, and writes $\mathsf{SMR}_0[i]$ and sources one $\mathsf{SMR}_0[j]$ and one $\mathsf{GPR}[k < 16]$ then the operation is performed as normal, but the non-existent shares of the $\mathsf{GPR}[k < 16]$ register are treated as zero. This allows for simple addition of round constants, which do not need to be masked, or rotations and shifts by values held in a register.
- An ALU instruction may not source an $\mathsf{SMR}_0[j]$ and write a $\mathsf{GPR}[k < 16]$. This will raise an Invalid Opcode Exception. Register-Immediate variants of add/sub, when sourcing an $\mathsf{SMR}_0[j]$, will also raise an Invalid Opcode Exception.
- When ALU instructions source and write to $\mathsf{GPR}[k < 16]$, they behave as normal. This enables standard address arithmetic etc.
- Arithmetic instructions `add, sub` are polymorphic with smectl.T. Depending on the value of smectl.T when executed, they perform their respective operations using boolean or arithmetic masked representations of their input variables. This allows SME to be used for algorithms favouring boolean masking, such as block ciphers, or arithmetic masking, such as some public key cryptography.
- Arithmetic instructions `mul, mulh, mulhu, mulhsu` only support arithmetic masking (smectl.T = 0). Executing any of these instructions on an $\mathsf{SMR}_w[h]$ en smectl.T = 1 results in an Invalid Opcode Exception.

When smectl.D is zero, the base ISA instructions work as normal, regardless of which registers they address. The following ALU instructions work with the SME extension: `add, sub, srli, slli, xor, or, and, rori, mul, clmul, grevi, shfli`. We also note that more specialist instructions from the scalar cryptography ISE for SHA2, SM3, SM4 and AES acceleration can also be made to work with this scheme, which we demonstrate later.

The polymorphism of the arithmetic instructions based on smectl.T requires some justification. We expect algorithms which require mixed arithmetic and boolean operations

to pick boolean masking, and stick with it. The main use case for this mechanism is switching between algorithms dominated by one type of masking or another.

3.3 New Instructions

SME introduces a small number of new instructions for managing the additional state, and creating, refreshing and destroying shared variables.

3.3.1 Loading and Storing Shares

Loading and storing of shares is an expensive part of any masked software implementation in terms of energy consumption, performance, static code size and dynamic instruction fetch bandwidth. It also represents a serious risk to security, where undocumented registers in the memory data paths cause collisions between shares. This problem increases with the number of shares, so providing good architectural support for it is essential. Here, we detail several mutually complementary approaches to this problem. Figure 3 gives pseudo code for each of the various approaches.

Given the design outlined in Section 3.1, there is some overhead in switching between share banks to do loads and stores. This cost is only paid if the software is truly generic and supported hardware security order independent. For software written specifically for $N \leq \mathsf{SMAX}$ shares, this cost can be removed.

We note that the bank select mechanism rewards programmers who load and store all the i'th shares of N variables, followed by all $i+1$'th shares of N variables. This implicitly avoids share collisions caused by sequentially loading and storing all shares of variable N, followed by all shares of variable M. This is because the total number of instructions is minimised by minimising the number of changes to the bank select field. Give that accidentally storing shares of the same variable sequentially is a well documented "gotcha" for side-channel engineers [CGD18, SSB+19, MPW21] we believe that providing a performance incentive for not doing this is a nice way to minimise the likelihood of it occurring.

The following sub-sections explain the different kinds of load and store instructions we considered including in SME, and finish which the rationale for what we actually included. Pseudo-code for each instruction variant is found in Figure 3.

Single share load and store. This is the simplest approach, where we overload the existing RISC-V load and store instructions to load and store shares between SME registers and memory. The base load word/half/byte instructions will load data from memory into the $\mathsf{SMR}_{\mathsf{smectl.B}}[x]$ when $x \geq 16$ (remembering that $\mathsf{SMR}_0[16 \leq x \leq 31]$ aliases the GPRs) and $\mathsf{GPR}[x]$ when $x \leq 15$. Likewise, store instructions can write the value $\mathsf{SMR}_{\mathsf{smectl.B}}[x]$ when $x \geq 16$ and $\mathsf{GPR}[x]$ when $x \leq 15$.

This approach requires the programmer to carefully schedule instructions to avoid share collisions. Though they are incentivised to avoid this when loading multiple variables by the bank switching mechanism, loading or storing a single shared variable will still result in share collisions when the instructions are executed back to back. This would require the implementation to transparently flush registers when load and store instructions operate on SME registers. Additionally, this approach causes an overhead in terms of instruction execution counts proportional to the number of shares. Note the code size *need not* grow with the number of shares, since we can loop over values of smectl.B rather than explicitly loading and storing registers from each bank.

Single variable load and store. This approach also overloads the base ISA load and store instructions. However, instead of loading or storing a single share of variable V in $\mathsf{SMR}_{i \leq \mathsf{smectl.B}}[V]$, we have the instructions load or store all shares of the variable in a single

`lw rd, imm(rs1)`:
→ if rd < 16 or !smectl.D: GPR[rd] $\leftarrow$ MEM[GPR[$rs1$] + imm]
→ else: $\mathsf{SMR}_{\mathsf{smectl.B}}[rd] \leftarrow \mathsf{MEM}[\mathsf{GPR}[rs1] + imm]$

(a) Functionality of the load single share instructions, which overload the base RISC-V load/store word instructions when smectl.D $\neq 0$

`sme.lv rd, stride(rs1)`:
→ for i = 0..smectl.D: $\mathsf{SMR}_i[rd] \leftarrow \mathsf{MEM}[\mathsf{GPR}[rs1] + stride \times i]$

(b) Functionality of the **l**oad whole **v**ariable instruction.

`sme.lis {rdx,..rdy}, stride(rs1)`:
→ for i = rdx..rdy: $\mathsf{SMR}_{\mathsf{smectl.B}}[i] \leftarrow \mathsf{MEM}[\mathsf{GPR}[rs1] + stride \times i]$

(c) Functionality of the **l**oad i'th **s**hares instruction.

`sme.lmv {rdx,..rdy}, stride(rs1)`:
→ for i = rdx..rdy:
→→ for j=0..smectl.B: $\mathsf{SMR}_{\mathsf{smectl.j}}[i] \leftarrow \mathsf{MEM}[\mathsf{GPR}[rs1] + \mathsf{smectl.B} \times j + stride \times i]$

(d) Functionality of the **l**oad **m**ultiple **v**ariables instruction.

Figure 3: Pseudo-code showing the functionality of the different load share instructions. Store instructions are defined the same way but wit the direction of the memory/register assignment reversed.

instruction. If the set of SME banks is viewed as a matrix where each bank is a column and each variable is a row, these instructions work row-wise across variables. This is very similar to the instructions in the base RISC-V vector extension which load and store an entire vector register.

This means that for the programmer, the instruction sequence for loading an un-masked implementation is identical to a masked implementation. They never need to query the value of smectl.D, or change the value of smectl.B, since the instruction can loop over the relevant shares automatically. It also means the instruction memory accesses are the same regardless of the number of shares, resulting in an energy, static code size and instruction count improvement. A performance improvement is possible based on the exact implementation, but not guaranteed.

In this case, the implementation (rather than the programmer) is responsible for making sure that shares do not collide during loading and storing. This might involve flushing the data path with dummy operations or random data. This is perfect in terms of making the SME implementation for maintaining security (and so freeing the programmer of the burden), though it does make the implementation much more complex. This complexity may be disproportionate considering the implications it has for the wider CPU sub-system and SoC system.

Multiple share load and store. Here, rather than overloading the existing load and store instructions, we introduce new, dedicated instructions for loading and storing the i'th shares of a set of multiple variables. For example, we load or store $\mathsf{SMR}_{\mathsf{smectl.B}}[i]\ \forall i \in 16..31$. Looping over values of smectl.B allows us to store entire variables of smectl.D shares. When smectl.D $= 0$, these instructions act like base ISA load/store instructions, i.e. they operate as if smectl.D $= 1$. Continuing the matrix analogy for the SME state, these instructions work column-wise across banks.

This sort of instruction can be thought of as fusing together multiple single share load and store instructions. It forces programmers to operate on all i'th shares, then all $i + 1$'th shares, and so implicitly guards against manipulating shares of the same variable sequentially, causing security problems. It also provides a code size and instruction count benefit.

However, in the case where we only want to (for example) spill a single variable to

the stack, we still need to iterate over every share and load/store it individually again, which considerably increases the code size and instruction execution counts over the single variable load and store instructions. Further, there is some complexity in how to select which registers to be loaded or stored. E.g. should the instruction allow only contiguous registers (e.g. $16, 17, 18$) to be loaded and stored, or allow any arbitrary list (e.g. $16, 19, 22$). The former is easier to implement, the latter makes register allocation simpler. In either case, the exact method of delivering the operand is also an open question. An additional register input could be used, or an immediate.

Multiple variable load and store. These instructions allow all shares of multiple variables to be loaded and stored in a single instruction. They allow the implementation to easily order loads and stores such that corresponding shares of the same variable do not interact when loading or storing multiple variables. They are very similar to the base RISC-V vector extension load/store whole register instructions, which allow multiple vector registers to be loaded or stored in a single instruction.

These instructions are maximally complex in their implementation, but give the largest benefit in terms of code size and executed instruction counts. There is also the problem of how to select which registers to load/store, as the the multiple share load/store instructions.

Included load/store instructions. After considering these various approaches and considering the implementation complexity, we opted only to include overloaded versions of the single share load/store instructions. Our rationale is:

- They require the least overhead in terms of additional logic, which keeps implementation complexity down. While this is useful in its own right, it is especially important for leakage resistant designs, since any additional complexity makes it harder to track separation of shares.
- They best fit with the "RISC" design principles of the host ISA. This has important implications for standards adoption, and indeed, whether any engineers will even want to implement the design.
- They give programmers the most flexibility in terms of scheduling instructions. In our experiments, we found that very rarely were we loading or storing just one variable at a time, meaning we could make sure that shares of a corresponding variable were never accessed adjacently.
- Other instruction variants which stored multiple shares of the same variable had this problem hard-coded into their design. We believe avoiding codifying such implementation difficulties into the ISA is essential.

While the power, performance and code size impact of our choice is detrimental, we think it is acceptable to *spend* some of the improvements gained through more efficient masked computations in keeping the rest of the design simple and easy to implement. Indeed, our final evaluations still came out smaller and faster than either pure-software masked implementations, or ISE assisted ones.

3.3.2 Masking Operations

We also introduce two new instructions for en-masking and re-masking values. These are very similar to those introduced in [GGM+20], but adapted to work on multiple shares.

`sme.mask rd, rs1` - Given source register $\mathsf{GPR}[rs1]$, create an $\mathsf{smectl.D}$-share boolean or arithmetic masked representation (according to $\mathsf{smectl.T}$), writing the result shares to $\mathsf{SMR}_i[rd]$ $\forall i \in \{0..\mathsf{smectl.D}\}$.

`sme.remask rd, rs1` - Given source register $\mathsf{SMR}_i[rs1]$, re-mask each share using fresh masks, writing the result shares to $\mathsf{SMR}_i[rd]$ $\forall i \in \{0..\mathsf{smectl.D}\}$.

Unlike [GGM+20], we deliberately omit an `sme.unmask` instruction. Any un-masking logic in a system must be well isolated from the main masked data-path so it does not cause un-intended un-masking of values during normal operation. For a 2-share designs, this is fairly simple, but the complexity grows considerably for higher order schemes. Given that un-masking is inherently not a secure operation, we believe the complexity and logic cost is wasted by supporting this in hardware. While there is a performance cost associated with doing un-masking in software, this is more than made up for by the performance gains SME provides to the compute part of masked operations.

3.4 Verification of Security Guarantees: A discussion

Power side-channels destroy every abstraction boundary that computer science and micro-electronics / VLSI design has erected in order to manage the complexity of building hardware and software systems. They are a caused by interactions at every point in the abstraction stack, making it extremely difficult to express security guarantees about them at any single point in the stack. Contrast this with timing channels, which can be effectively mitigated at several points in the abstraction stack (with varying levels of performance implications), and which exist only because software has access to accurate timers. We cannot simply take away the observability of power consumption.

As such, the following guarantees we want SME to provide should serve as the *starting point* for a discussion around what sorts of guarantees an ISA *can* and *should* make about power side-channel security. Are they strong enough to support formal reasoning? Are they enough to make secure implementations portable between different devices implementing SME? Are they possible to evaluate, verify or falsify? Are the requirements they place on implementations weak enough to make their realisation feasible, both in terms of complexity, cost, and *commercial or standardisation viability*?

Going forward, we define an "SME Instruction" as one which executes when $\mathsf{smectl.D} > 0$, and which reads and writes at-least one SME register. We suggest this list of security claims each implementation of SME should provide:

1. When smectl.D is set to 0, the implementation provides no guarantees of side-channel security.
2. When smectl.D is set to $i > 0$, the implementation provides some level of security guarantee, which is a function of the number of shares being used. Again, this accounts for decoupling the number of shares used from the architecture of SME. The platform or software environment should provide a mechanism for discovering the security order based on smectl.D. Going forward, we refer to the security order provided by an implementation as O(smectl.D).
3. The SME implementation *must not* cause any intra-instruction leakage which is visible to O(smectl.D) analysis. That is, the implementation of the instructions themselves must not cause a security order reduction. This can be verified by executing SME instructions in isolation and using a TVLA flow to check for leakage.
4. The SME implementation *must not* cause inter-instruction leakage between N-adjacent SME instructions. That is, instruction i will not interact with any other instructions within N instructions of it in a way which undermines O(smectl.D) security. N will be guaranteed by the implementation, and will likely be dependent on the length of any processing pipelines.
5. The SME implementation *must* document when it does not flush registers which contain sensitive values.
6. The SME implementation *does not* guarantee any security order if the programmer explicitly manipulates shares outside the other SME guarantees. That is, SME is not impervious to willful misuse, and will not trap or cause an exception if this happens. For comparison, a programmer can willfully cause their program to segfault, causing

the processor to catch the invalid access and stop it to prevent a security hole. No such guarantee or safety net exists for deliberately undermining the security of SME.

We hope that this list is necessary and sufficient, but welcome feedback to the contrary, or ways to better express the security requirements. In terms of *verifying* these claims an implementation must make, there are several overlapping considerations.

Side-channel based security guarantees resist standard methods for testing or certifying compliance with an ISAs functional specification, simply because they make claims about things (power consumption) which are traditionally far outside the remit of the ISA to specify. We belive that the *functional* aspects of SME are no more or less challenging to verify than any other comparable architectural extension. Indeed, they are well suited to any co-simulation, constrained random stimulus (CRS) based verification methodology as used in industry, or even model-checking based verification. One difficulty is the inherently non-deterministic result values of some instructions. In a CRS verification environment, this can be handled by hinting to the golden reference model what the actual values are from the hardware, which is a well established method for handling permissable non-determinism in a design. For formal functional verification, one may write assertions such that the combination of share values always represent the correct value.

For verifying the *security* aspects of SME, the job of the verification engineer is considerably harder. We note that there are standard evaluation techniques for devices seeking certification for side-channel security, which implementers may aim for even before fabricating a chip. E.g. ISO 17825, though this particular standard is not without shortcomings [WO19]. There is also increasing literature in design-time tooling for side-channel security, which is well summarised in [BBYS21]. One promising approach is the one described in [GHP+21], where the authors are able to co-verify a software masked implementation inside the CPU hardware which executes it. We believe this is an excellent way to approach the security verification of ISA-assisted masking. Our point overall is to note the inherent challenge of ISA based side-channel security, and that there are tools and standards which tame this problem to the point of feasibility.

3.5 Implementation Considerations

Here we consider various implementation choices for the SME architecture. Most of these considerations also exist in a standard vector processor, but with important intersections with security considerations. We acknowledge that most computer architects would not call our design a *vector processor* in the most accurate sense. We believe however that there are enough similarities to make it a useful tool for explaining the concepts of SME, and as a comparison point.

The first obvious choice for implementers is to decide on the maximum number of shares supported by their hardware design. We include support for upto 16 shares in the smectl.r egister, which we felt was more than sufficient[3] for any reasonable implementation.

As a guide for how many shares are "affordable", we use the cost of an ECC accelerator and an AES accelerator as a rough guide. An un-masked ECC accelerator takes $2-5K$[4] FFs to store its state for certain parameter sizes. An unmasked AES accelerator supporting upto AES-256, requires 128 FFs for the state, and 256 FFs for the key, at minimum, before intermediate processing registers are accounted for. For an un-masked pair of accelerators, this totals 2.5-5.5K FFs. For an SMAX-share implementation of SME, we need $\mathsf{XLEN} * (\mathsf{SMAX} - 1) * 16$ *additional* register bits, on top of the $31 * \mathsf{XLEN}$ general purpose register file. Again, ignoring registers needed for intermediate processing, For XLEN =32, we could have SMAX =4 and still save architectural registers. Of-course, we

[3] At the risk of falling into the "X should be enough for anyone" trap, so often evident in the history of computer engineering.

[4] https://eprint.iacr.org/2020/1148.pdf - Table III.

cannot expect SME to achieve the same level of performance, but this is the trade-off SME offers, flexibility and size at the cost of raw speed.

Given a number of hardware shares, the next decision is the width of the data-path for the shared operations. For linear operations where shares do not need to interact, this can be dictated by how many instances of the linear operations can be implemented in parallel. For example, in a 4 share design, the implementer might only instance a single ALU containing xor/not/shift/rotate etc, and process each share sequentially (with appropriate barriers between shares to prevent leakage), with an obvious saving of silicon area at the cost of performance. Alternatively, a high performance implementation might build one linear operation per share, and execute them all in parallel, sacrificing silicon area for performance gain.

Care must be taken with the sequential processing approach not to overwrite shares and cause a security order reduction in the micro-architecture. This can be solved easily by transparently adding and removing temporary masks pre/post instruction execution.

For non-linear operations (AND, SBoxes) where shares must interact, we note that implementers could vary the width of the data-path to trade off between area and performance. For example, they could process the low 16 bits of a bitwise AND, followed by the high 16 bits over two cycles.

4 Example Hardware Implementation

Here we describe an example implementation of SME. We emphasise that this is *an* implementation, and has been constructed as a proof of concept. A large space of possible implementations exist, and the main focus of this work is the *architecture* of SME. This proof of concept is only to demonstrate it's feasibility.

Our implementation is based on the free and open source SCARV-CPU core. It is a 5-stage pipelined, in order, micro-controller class CPU. The 5 stages are fetch, decode, execute, memory and write-back. A micro-architecture block diagram, showing the SME enhanced core is show in Figure 4.

Domain Oriented Masking (DOM) [GMK16] is used to guarantee side-channel security for all of the SME operations. This was chosen because it scales well to higher masking orders and so acts as a good low level primitive for SME to expose at the ISA level. It is also able to provide formal guarantees of security under certain assumptions. This makes our implementation amenable to the kind of co-verification described in [GHP+21].

Our hardware is designed such that with a single Verilog parameter change, we can increase or decrease the masking order of the design. We do this by generating the required number of SME Bank registers, shared data-paths and DOM gadgets at synthesis time. We use this later to show that identical code can automatically take advantage of higher masking orders, as supported by the underlying hardware.

4.1 Data-path Description

For load instructions, when SME is off, load instructions behave as normal. When SME is on, and the loaded data is sent to either the GPRs if the register address is < 16, or to the i'th register of the currently selected SME Bank. For store instructions, the register write data is sourced from either the GPRs in the decode stage, or later, the SME Bank register is sourced in the Execute stage, before the memory transaction is started in the Memory stage.

The SME compute instructions implemented as an additional execute stage functional unit. If SME is turned off, then the normal ALU handles the operations, as all of the operands come from the GPRs via pipeline registers `s2_opr_a/b`. If SME is turned on, and the instruction is an SME instruction, then upto two shares (of different variables)

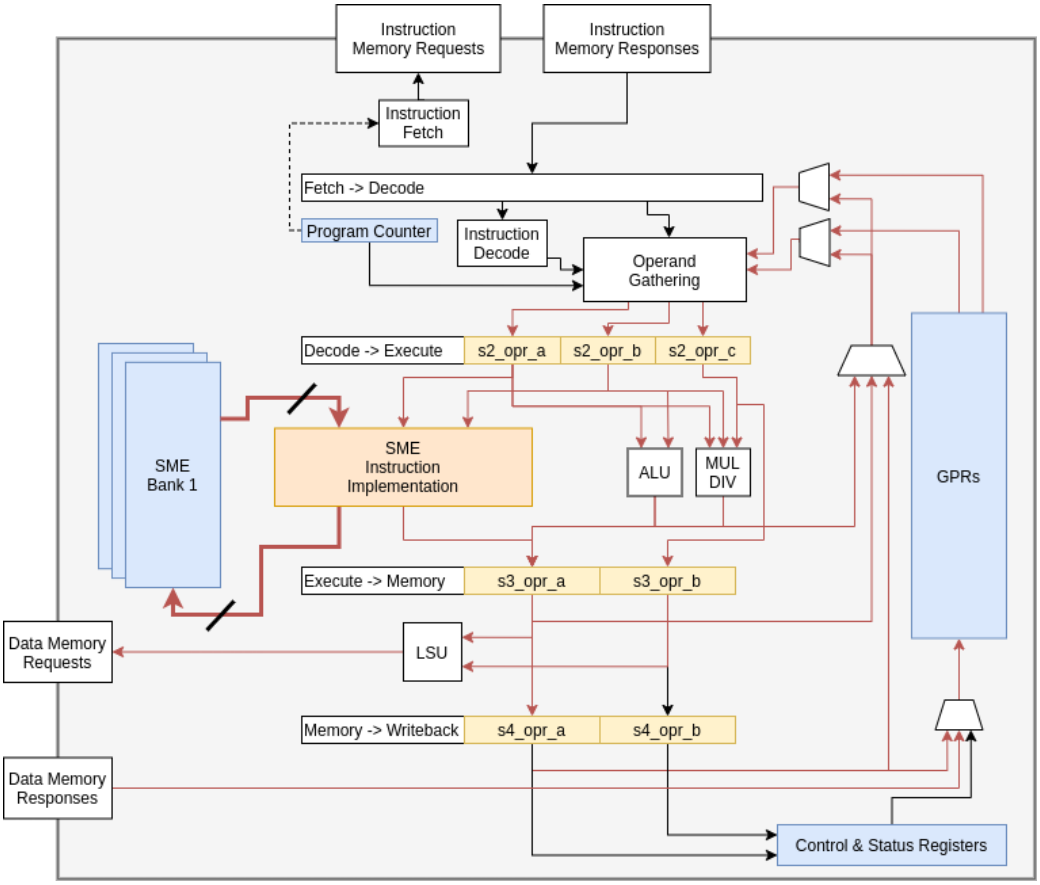

Figure 4: Micro-architecture diagram of the modified SCARV-CPU with the SME extensions implemented.

are fed to the SME functional unit via the `s2_opr_a/b` pipeline registers. The remaining shares are sourced from the SME Bank registers. Other than for store instructions, shares from SME Bank Registers cannot enter the main execution pipeline, making reasoning about share collisions a much smaller problem. We discuss this further in a later section.

To show that SME lends itself to small, embedded class cores which are most susceptible to side-channel attack, we opt for an area optimised design. To this end, the SME data-path from the Bank register read ports, through the execution units to the Bank register write ports is single cycle (baring any multi-cycle instructions described later), with no additional pipelining. While this adds a penalty in terms of maximum operating frequency, it adds a considerable saving in terms of core area, and as described in the evaluation section, the core is still able to run at a reasonable frequency even in an FPGA. Further, the lack of micro-architectural registers makes reasoning about possible sources of leakage much simpler.

4.1.1 Randomness Generation

To generate the fresh randomness used by the SME instructions, we use a single round of the Keccak permutation [BDPVA13]. The exact parameterisation of the permutation is chosen to be the smallest one such that the state size is large enough to provide enough random bits for the current SMAX value. Keccak was chosen for its very good mixing of the state even after a single round. The Keccak state is seeded with a constant at reset, but continually re-seeded every cycle when SME is turned on. The additional entropy is

sourced from a single bit of a ring-oscillator based TRNG, which is mixed into the state at every cycle.

While secure sourcing of random numbers is essential for masking, it is not the primary focus of this work. Other design questions include whether it would have been more efficient to use many instances of very small Keccak permutations, and what security tradeoffs this might bring. Other constructions could also be investigated other than Keccak.

We have listed this topic under "Implementation Considerations", though if a design like this were to be standardised, it is like that a minimum rate and quality of entropy for the mask generation would need to be defined. This would make it an *architectural* concern, so that software could rely on it.

4.1.2 Non-linear Base ISA Instructions

To implement the `and` instruction, we use the standard DOM AND gate implementation from [GMK16]. The `add` instruction is implemented using an iterative Kogge-Stone adder, as described in [GGM+20]. See Table 1 for how the randomness requirements of the `and` and `add` instructions scales with SMAX.

4.1.3 AES Instruction Implementation

To show that SME can handle instructions beyond the simple ones in the base ISA, we also implement the RISC-V `Zkne` extension instructions for accelerating AES encryption: `aes32es` and `aes32esm`. These instructions were specifically chosen to be lightweight to implement and be feasible for small cores to include. A full explanation of these instructions can be found in the draft RISC-V Cryptography ISE specifications [ZGS+21], and the associated paper [MNP+20]. We include a short description here for completeness.

They are based on a *T-tables in hardware* approach, where a single byte from one register operand, and the entire Sub-bytes, Shift-rows and Mix-columns steps are applied to this byte. In software, this would be done by performing a lookup into memory. The instructions wrap this lookup into a single instruction, and dynamically calculate the steps with each execution. The final step is to XOR the lookup result into the second register operand, which effectively performs the AddRoundKey step.

For our masked design, the key concern is the single SBox operation the instruction performs, as this is the only non-linear part of the computation. We created a naive N-order masked implementation of the AES Forward SBox[5]. The SBox takes 4 cycles to execute as the data propagates through the circuit. During this time, the rest of the CPU stalls, similarly to how it does for a multi-cycle multiply operation. While it would be possible to pipeline the instructions to hide this additional latency, this would require multiple 32-bit registers to pipeline the second register operand as well, which would be disproportionately expensive in the context of the wider system.

We note that the software using these AES instructions only ever uses 8 GPRs to store secret data. This raises the possibility of custom implementations of just these AES instructions, with only 8 sets of shared registers rather than the full SME complement of 16. We believe this could be a popular way for vendors to add side-channel security to these instructions.

4.2 Instruction Latencies

SME instruction latencies are shown in Table 1. Linear operations are single cycle. Non-linear operations are multi-cycle, but we utilise the very small combinatorial path depth

[5] Again, we treat the problem of an optimal N-order masked AES SBox as an important but separate problem from the one we are really trying to address in this paper.

Table 1: Table of instruction latencies and randomness consumption for SME instructions. Latency is measured in clock cycles, from rising edge to rising edge.

Instruction	Latency Baseline	Latency Masked	Random Bits Consumed / cycle
`mask`	1	1	$\mathsf{SMAX} \times \mathsf{XLEN}$
`remask`	1	1	$\mathsf{SMAX} \times \mathsf{XLEN}$
`and`	1	1	$\mathsf{XLEN} \times \frac{\mathsf{SMAX}(\mathsf{SMAX}+1)}{2}$
`add/sub`	1	6	$\mathsf{XLEN} \times \frac{\mathsf{SMAX}(\mathsf{SMAX}+1)}{2}$
`aes32es[m]`	1	4	$34 \times \frac{\mathsf{SMAX}(\mathsf{SMAX}+1)}{2}$
Linear Instrs	1	1	0

Table 2: Table of hardware overheads for the baseline SCARV-CPU, and when enhanced with SME at various masking orders. Timing Slack is shown targeting 50MHz on an FPGA. Negative Timing Slack implies the design does not meet the 50MHz frequency requirement.

Core Variant	Timing Slack	FFs	LUTs
SCARV-CPU Baseline	9.480ns	1814	5576
SCARV-CPU + SME (SMAX =2)	3.324ns	2714	8618
SCARV-CPU + SME (SMAX =3)	2.239ns	3437	10339
SCARV-CPU + SME (SMAX =4)	2.287ns	4716	12441

between DOM masking stages to reduce the latency by clocking subsequent registers on positive and negative edges of the clock. Hence, the AND operation which would normally take two cycles, now effectively takes one. The same is done for the AES instructions.

4.3 Overheads

To measure the overhead in a real system, we implemented the modified SCARV-CPU with SME in a SASEBO-GIII [HKSS12] side-channel analysis platform. The platform contains two FPGAs: a Xilinx Kintex-7 (model `xc7k160tfbg676`) target FPGA, and a Xilinx Spartan-6 (model `xc6slx45`) support FPGA. Our design uses only the former, and was implemented using Xilinx Vivado 2019.2. We used the default synthesis settings, with no extra effort put into post-synthesis or post-layout optimisation.

The design runs at 50 MHz in a single clock domain, driven from a down-scaled 200 MHz differential external clock.

Table 2 describes the overhead of SME on the SCARV-CPU for different masking orders. Note that we include these multiple masking orders to show approximately how our design scales, but have not yet evaluated that our specific design is secure for every masking order shown.

5 Software Performance Evaluation

This section describes our experiences writing side-channel secure software using SME. We also describe our efforts to empirically gain confidence in the side-channel security of our design.

To test real world cryptographic workloads using SME, we chose The AES-128 Key Schedule and Block Encryption, plus the ChaCha20 [Ber08] round function. These were chosen for their popularity in real-world deployments, in academic literature, and because they are very different primitives in terms of their implementation. Using AES also allowed us to test our side-channel secure implementation of the RISC-V Scalar Cryptography Instruction Set Extensions for AES.

All algorithms were checked for correctness and their performance for a range of SMAX values, representing different designs with increasing side-channel security levels, with results shown in Table 3.

Table 3: Table of software performance figures for different variants of SMAX.

	AES KS		AES Enc		ChaCha20	
Core Variant	Cycles	Inst	Cycles	Inst	Cycles	Inst
SCARV-CPU Baseline	378	222	296	241	1275	1068
SCARV-CPU + SME (SMAX =2)	769	382	1142	454	3071	1231
SCARV-CPU + SME (SMAX =3)	978	470	1333	549	3224	1323
SCARV-CPU + SME (SMAX =4)	1187	558	1524	644	3389	1415

For the AES Key-Schedule and Block Encryption, we had no struggles at all implementing the algorithms in the SME paradigm. The actual round functions remained identical to the standard assembly code. Our only modifications were to how the secret state was loaded into the CPU. This was done using a loop, to accommodate all values of SME using the same software. A typical example this loop is shown in Figure 5 for reference, as it is a common pattern in SME based software. This loop adds a small performance and code-size overhead due to the loop management instructions, but this is small compared to the *necessary* overhead due to loading in SMAX-times the amount of data for secret values.

For ChaCha20, again there were no major problems with the round function, which again, is identical to the un-protected variant. The 16 state words of ChaCha20 fit exactly into the 16 SME register banks. However, ChaCha20 does show the limitations of SME at the end of the algorithm, where the original state must be loaded back into the CPU and arithmetically added to the final round output. This requires us to push half of the final round state to the stack, load half the original state and add it to the final state, store this result, and then repeat for the other half of the state. For implementations with a high SMAX, this is potentially a very slow operation. However, our experiments showed it is still a small part of the runtime of the overall algorithm, which is still much faster for the hardware support offered by SME.

Comparing to [GGM+20] our design is almost identical in speed for AES-128 block encryption in terms of *cycles* (1113 v.s. 1142), but executes fewer than half the number of instructions (1012 v.s. 454) owing to the use of the RISC-V scalar cryptography extensions for AES. This alone would likely result in a considerable power and energy efficiency improvement. We also note our design could use pipelined implementations of the AES instructions (at a cost of hardware resources) to improve the cycle counts. We emphasise that allowing for this power/performance/area tradeoff while keeping software compatability is a key benefit of SME. For ChaCha20, our design is faster than [GGM+20] for both instruction and cycle counts by 1.26× and 1.2× respectively. Further, our architecture scales to higher masking orders while [GGM+20] does not.

Comparing to [GPSS18], which uses the ARM Neon SIMD extensions to implement a 4-share masked AES, we can see how beneficial the lightweight RISC-V AES instructions can be. Their implementation takes $12.3K$ clock-cycles or 474 cycles per byte when using pre-loaded randomness for re-sharing operations. Our implementation enables a 4-share AES-128 encryption to be executed in 1524 cycles, or 95.25 cycles per byte. We must note that this is not the most direct comparison, since the ARM Neon implementation they use has no dedicated instructions for AES, and relies on an external source of randomness. It does illustrate how much more efficient a RISC-V based design can be. We also note that ARM Neon designs come with no side-channel guarantees in their implementation (this is not a problem they are designed to solve), which is something SME tries to address directly.

6 On the Standard RISC-V Vector Extension

Though the key concept of our design is inspired by vector programming principles, we accept that from a computer architecture perspective, ours is not *strictly* a generic vector

```
li    t0, 0x1E0        // Constant to select highest supported SMAX
csrrw x0, smectrl, t0  // Select 0th bank, highest SMAX
csrr  t1, smectrl      // Read back SMAX and extract the maximum
srli  t1, t0, 5        // bank number. Used as loop end variable.
or    t1, t1, t0       // Create the loop start variable.
load_sme_state:
  csrrw x0, smectrl, t0 // Select i'th SME register bank
  lw    x16, 0*4(a1)    //
  lw    x17, 1*4(a1)    // Load 4 words of shared state and
  lw    x18, 2*4(a1)    // increment base register by size
  lw    x19, 3*4(a1)    // of unmasked state.
  addi  a1, a1, 44*4    //
  addi  t0, t0, 1       // Increment the bank register
  bne   t0, t1, load_sme_state
```

Figure 5: RISC-V assembly code, showing how to loop over the shares of a state array and load them into the SME registers. This specific example is for AES. The code for ChaCha20 is identical, but we load 16-words per loop iteration rather than 4. Storing the shares to memory is identical, but the `lw` instructions are simply replaced with `sw` instructions. We note the overhead of deriving the loop counter and end variables. This is amortised somewhat, as they are re-used for other subsequent loops (e.g. to store the state back to memory) and can be kept in registers to avoid re-deriving them. This was a benefit of using only half of the RISC-V GPRs to store mask shares, as the others are free for these sorts of control path variables.

processor. This comparison to vector programming is useful for explaining the concept, but there are some key tradeoffs we have made which are un-reasonable or non-sensical for more general purpose vector processing extensions, like the standard RISC-V Vector Extension (RVV). Here, we discuss these differences in approach, and how our design could be retrofitted onto RVV.

Data-path interactions. In RVV, very few vector instructions interact with the main general purpose registers, meaning they can potentially be executed in parallel. This is impossible in SME, because we re-use a portion of the GPRs to store shares. This is undesirable in a general purpose vector engine, since enabling instruction level parallelism leads to a performance gain. Our expectation is that this aspect of our design would be most distasteful to a computer architect. Though, we believe it is reasonable given our design goals, because we are optimising for security and then performance.

Number of registers. The major difference between SME and RVV is the choice of architectural registers. RVV adds 32 separate vector registers on top of the 32 base RISC-V GPRs, where each vector register is a power-of-two $>$ XLEN bits long. SME in contrast adds SMAX $-$ 1 16-entry register files, each of which are XLEN bits wide, and uses the top 16 existing GPRs to represent the 0'th share.

For RVV, this makes perfect sense as a general purpose vector engine. The choice for SME was motivated by the supposition that, given 32 extra XLEN registers were going to be added, from a security perspective it is better to support 16 3-share variables in hardware, than 32 2-share variables. E.g. some must hold loop counters, stack pointers, return addresses and array pointers, none of which need masking. Essentially, it allowed us to represent the most variables with the highest security order for the minimum number of additional architectural registers.

Choice of instructions. For SME, we include only the instructions most useful for implementing (masked) cryptographic code, which is obviously a much smaller set than is supported by the more general purpose RVV. The omission of most instructions from SME can be explained simply by the fact they are un-necessary.

However, there are some instructions in RVV which are *intrinsically dangerous* in the context of hardware masking support through an ISE. These can be broadly described as instructions which allow data to move laterally across a vector register. For example, the `slideup` and `slidedown` instructions, which move all vector elements one to the left or right. These are a *built in* danger to masking schemes, because they implement functionality which enables/forces shares to collide and thus undermine the security.

In SME, there *is no architectural way* to overwrite one SME share with another. No instruction exists which takes a value from $\mathsf{SMR}_i[x]$ and moves it to $\mathsf{SMR}_{j\neq i}[x]$. The only way to do this is for the programmer to explicitly do this by moving values to memory and back again, which we consider a reasonable indicator that this is a bad idea. We believe this is a clear advantage over designs such as [GGM+20], where the programmer must be very careful not to accidentally combine shares, owing to it being supported explicitly in the architecture.

Flexible Vector Lengths. RVV is extremely capable at supporting variable length vectors, and comes with some necessary additional complexity to support this. This includes variable lengths of vector, with different data types. For example, it supports N length vectors of bytes or words natively.

SME, in contrast, is designed only to work with short vectors upto smectl.D in length, which are made up of XLEN wide elements. Again, this is simply because it focuses only on the masked cryptography use case.

Possible Extensions to RVV. Having described SME, it is useful to consider how RVV *could* be extended with similar support for hardware assisted software masking countermeasures.

Architecturally speaking, we believe the most useful instructions to add from a programmers perspective are masked bitwise non-linear logic instructions: `and`/`or`. This is because they are the most difficult and expensive operations to implement securely in pure software. However, these are also the most awkward to add from a hardware perspective, because (assuming shares are represented laterally across the vector registers) they require all shares to be combined to compute the result, rather than the result being computable share-wise as with linear operations like `xor`. RVV already well supports instructions for linear operations which operate a share at a time. Also, RVV would benefit from masking and re-masking instructions as described in Section 3.3.2.

In this sense, we believe that a modified vector programming model is a compelling one for supporting higher order-masked cryptographic implementations at the ISA level. However, the main difficulty clearly lies in upholding the assumptions on which masking security relies in the implementation, as discussed in Section 3.4. This is particularly hard in a more general purpose vector engine like RVV, which must balance the security requirements described here with wanting to be as performant as possible. For example, the need to flush registers to guarantee security will often have a performance cost. Recent work analysing how high-performance micro-architectures ruin the security of carefully crafted side-channel secure code [GPM21] shows this is clearly an interesting open research problem.

A compromise is to simply accept that a D-share implementation will not achieve the maximum level of security theoretically possible with D shares. The implementation must simply report a lesser value of O(smectl.D). Critically, this still allows for *some guarantee* of security, while still providing maximal performance for the general purpose use cases.

Overall, we believe this comparison to RVV shows just how special purpose software

based masking countermeasures are as a workload. As architectures like RISC-V allow for more and more domain specific processor designs, we believe this sort of hardware-software co-design will become much more common. Though, the impact of modifying a generic core for a specific workload must still be weighed against how affects the efficiency of other operations the core must perform.

7 Conclusion

We have described an Instruction Set Extension for supporting power and EM based side-channel security using higher order masking based countermeasures. The ISE allows the same software to benefit from the maximum security order supported by the underlying hardware, which restores a key promise of ISAs: that improvements to underlying hardware (both in terms of performance and security) are automatically passed on to software for free. The key concept being to give software visibility of how many hardware shares are supported, allowing generic code to be written supporting however many shares the hardware does.

Our design works well for popular cryptographic workloads, and is suitable not just for very simple "base" ISA instructions, but also for lightweight cryptographic instructions proposed for the RISC-V ISA. We believe this is important, because it shows our architecture works well as a *framework* for managing the additional state required by masking countermeasures. This means it can compose with other cryptographic instructions defined in the future, perhaps which target Post-Quantum Cryptographic workloads, or schemes from the NIST Lightweight Cryptography competition.

We recognise that our evaluations so far only focus on showing how the hardware and software remains functionally correct and performant across multiple masking orders. Our current and future work is focusing on demonstrating the actual side-channel security of the system.

Future research is also needed on whether or not the ISA is indeed the right place to put side-channel security guarantees in the hardware/software stack. While our work shows this is possible and efficient as a real-world solution, a framework for objective comparisons of the pros and cons of where security guarantees exist in the hardware/software stack, and the implications for side-channel security is still needed.

We hope our implementation acts as a platform or data point which others can use as a basis for this sort of research.

Acknowledgements

This work-in-progress has been supported in part by EPSRC via grant EP/R012288/1, under the RISE (http://www.ukrise.org) programme.

References

[BBYS21] Ileana Buhan, Lejla Batina, Yuval Yarom, and Patrick Schaumont. Sok: Design tools for side-channel-aware implementions. *arXiv preprint arXiv:2104.08593*, 2021.

[BDPVA13] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 313–314. Springer, 2013.

[Ber08] D.J. Bernstein. ChaCha, a variant of Salsa20. 2008. https://cr.yp.to/chacha/chacha-20080120.pdf.

[CGD18] Y. Le Corre, J. Großschädl, and D. Dinu. Micro-architectural power simulator for leakage assessment of cryptographic software on ARM Cortex-M3 processors. In *Constructive Side-Channel Analysis and Secure Design (COSADE)*, LNCS 10815, pages 82–98. Springer-Verlag, 2018. https://doi.org/10.1007/978-3-319-89641-0_5.

[DMGH19] Elke De Mulder, Samatha Gummalla, and Michael Hutter. Protecting risc-v against side-channel attacks. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–4. IEEE, 2019.

[GGM+20] Si Gao, Johann Großschädl, Ben Marshall, Dan Page, Thinh Pham, and Francesco Regazzoni. An instruction set extension to support software-based masking. Cryptology ePrint Archive, Report 2020/773, 2020. https://eprint.iacr.org/2020/773.

[GHP+21] Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. Coco: Co-design and co-verification of masked software implementations on cpus. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[GJM+17] Hannes Gross, Manuel Jelinek, Stefan Mangard, Thomas Unterluggauer, and Mario Werner. Concealing secrets in embedded processors designs. In Kerstin Lemke-Rust and Michael Tunstall, editors, *Smart Card Research and Advanced Applications*, pages 89–104, Cham, 2017. Springer International Publishing.

[GMK16] Hannes Groß, Stefan Mangard, and Thomas Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. In *TIS@ CCS*, page 3, 2016.

[GMPP20] S. Gao, B. Marshall, D. Page, and T. Pham. FENL: an ISE to mitigate analogue micro-architectural leakage. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2020(2):73–98, 2020.

[GPM21] Barbara Gigerl, Robert Primas, and Stefan Mangard. Secure and efficient software masking on superscalar pipelined processors. Cryptology ePrint Archive, Report 2021/1110, 2021. https://ia.cr/2021/1110.

[GPSS18] Benjamin Grégoire, Kostas Papagiannopoulos, Peter Schwabe, and Ko Stoffelen. Vectorizing higher-order masking. In Junfeng Fan and Benedikt Gierlichs, editors, *Constructive Side-Channel Analysis and Secure Design*, pages 23–43, Cham, 2018. Springer International Publishing.

[HKSS12] Y. Hori, T. Katashita, A. Sasaki, and A. Satoh. SASEBO-GIII: A hardware security evaluation board equipped with a 28-nm FPGA. In *IEEE Global Conference on Consumer Electronics*, pages 657–660, 2012.

[KS20] Pantea Kiaei and Patrick Schaumont. Domain-oriented masked instruction set architecture for risc-v. *IACR Cryptol. ePrint Arch.*, 2020:465, 2020.

[MNP+20] Ben Marshall, G. Richard Newell, Dan Page, Markku-Juhani O. Saarinen, and Claire Wolf. The design of scalar aes instruction set extensions for risc-v. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):109–136, Dec. 2020.

[MPW21] Ben Marshall, Dan Page, and James Webb. Miracle: Micro-architectural leakage evaluation. Cryptology ePrint Archive, Report 2021/261, 2021. https://eprint.iacr.org/2021/261.

[ris] RISC-V vector specification. https://github.com/riscv/riscv-v-spec.

[SBB+17] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, et al. The arm scalable vector extension. *IEEE micro*, 37(2):26–39, 2017.

[SSB+19] Madura A Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. Rosita: Towards automatic elimination of power-analysis leakage in ciphers. *arXiv preprint arXiv:1912.05183*, 2019.

[WO19] Carolyn Whitnall and Elisabeth Oswald. A critical analysis of iso 17825 ('testing methods for the mitigation of non-invasive attack classes against cryptographic modules'). In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 256–284. Springer, 2019.

[ZGS+21] Alexander Zeh, Andy Glew, Barry Spinney, Ben Marshall, Daniel Page, Derek Atkins, Ken Dockser, Markku-Juhani O Saarinen, Nathan Menhorn, Richard Newell, et al. Risc-v cryptographic extension proposals volume i: Scalar & entropy source instructions. https://github.com/riscv/riscv-crypto, 2021.

A Example Software

Here we give an example of writing an AES-128 block encryption for generic masking orders, as supported by SME:

```
1  .global sme_aes128_enc_block
2  .func   sme_aes128_enc_block
3  sme_aes128_enc_block:
4      // a0 - uint32_t ct[4],
5      // a1 - uint32_t pt[4],
6      // a2 - uint32_t rk[SME_SMAX][AES128_RK_WORDS]
7      addi    sp, sp, -32
8      sw      x18, 0*4(sp)
9      sw      x19, 1*4(sp)
10     sw      x20, 2*4(sp)
11     sw      x21, 3*4(sp)
12     sw      x22, 4*4(sp)
13     sw      x23, 5*4(sp)
14
15
16     li t0, 0x1E0          // D=max, bank=0
17     csrrw x0, 0x006, t0   // Turn on SME with MAX shares and 0'th bank.
18     csrr  t1, 0x006
19     srli  t1, t1, 5       // t1 contains loop end var for share ld/st
20
21     or  t1, t1, t0        // Load 0'th round key.
22     mv  t2, a2
23     .aes128_enc_blk_ldkey0:
24         csrrw x0, 0x006, t0      // Select i'th bank
25         lw   x16, 0*4(t2)        // Load i'th shares
26         lw   x17, 1*4(t2)        // Load i'th shares
```

```
27          lw   x18, 2*4(t2)       // Load i'th shares
28          lw   x19, 3*4(t2)       // Load i'th shares
29          addi t2, t2, 44*4
30          addi t0, t0, 1
31          bne  t0, t1, .aes128_enc_blk_ldkey0
32
33      lw a3, 1*4(a1)
34      lw a4, 2*4(a1)
35      lw a5, 3*4(a1)
36      lw a1, 0*4(a1)          // Load plaintext. a1 now available.
37
38      xor x17, x17, a3
39      xor x18, x18, a4
40      xor x19, x19, a5
41      xor x16, x16, a1        // 0'th add round key.
42
43      mv a1, x0               // Clear plaintex from non SME registers.
44      mv a3, x0
45      mv a4, x0
46      mv a5, x0
47
48      addi a1, a2, 16*10          // Loop end pointer
49
50  .aes128_enc_blk_l0:
51
52      li  t0, 0x1E0               // D=max, bank=0
53      mv  t2, a2
54      .aes128_enc_blk_ldkey1:
55          csrrw x0, 0x006, t0     // Select i'th bank
56          lw   x20, 16+0*4(t2)    // Load i'th shares
57          lw   x21, 16+1*4(t2)    // Load i'th shares
58          lw   x22, 16+2*4(t2)    // Load i'th shares
59          lw   x23, 16+3*4(t2)    // Load i'th shares
60          addi t2, t2, 44*4
61          addi t0, t0, 1
62          bne  t0, t1, .aes128_enc_blk_ldkey1
63      xor x0,x0,x0; xor x0,x0,x0; xor x0,x0,x0;
64
65      aes32esmi x20, x20, x16, 0        // Even Round
66      aes32esmi x20, x20, x17, 1
67      aes32esmi x20, x20, x18, 2
68      aes32esmi x20, x20, x19, 3
69      aes32esmi x21, x21, x17, 0
70      aes32esmi x21, x21, x18, 1
71      aes32esmi x21, x21, x19, 2
72      aes32esmi x21, x21, x16, 3
73      aes32esmi x22, x22, x18, 0
74      aes32esmi x22, x22, x19, 1
75      aes32esmi x22, x22, x16, 2
76      aes32esmi x22, x22, x17, 3
77      aes32esmi x23, x23, x19, 0
78      aes32esmi x23, x23, x16, 1
```

```
79      aes32esmi x23, x23, x17, 2
80      aes32esmi x23, x23, x18, 3          // x20-x23 contains new state
81
82      li  t0, 0x1E0                 // D=max, bank=0
83      mv  t2, a2
84      .aes128_enc_blk_ldkey2:
85          csrrw x0, 0x006, t0       // Select i'th bank
86          lw   x16, 32+0*4(t2)      // Load i'th shares
87          lw   x17, 32+1*4(t2)      // Load i'th shares
88          lw   x18, 32+2*4(t2)      // Load i'th shares
89          lw   x19, 32+3*4(t2)      // Load i'th shares
90          addi t2, t2, 44*4
91          addi t0, t0, 1
92          bne  t0, t1, .aes128_enc_blk_ldkey2
93      xor x0,x0,x0; xor x0,x0,x0; xor x0,x0,x0;
94
95      addi a2, a2, 32               // Increment round key pointer.
96      beq  a1, a2, .aes128_enc_blk_final
97
98      aes32esmi x16, x16, x20, 0          // Odd Round
99      aes32esmi x16, x16, x21, 1
100     aes32esmi x16, x16, x22, 2
101     aes32esmi x16, x16, x23, 3
102     aes32esmi x17, x17, x21, 0
103     aes32esmi x17, x17, x22, 1
104     aes32esmi x17, x17, x23, 2
105     aes32esmi x17, x17, x20, 3
106     aes32esmi x18, x18, x22, 0
107     aes32esmi x18, x18, x23, 1
108     aes32esmi x18, x18, x20, 2
109     aes32esmi x18, x18, x21, 3
110     aes32esmi x19, x19, x23, 0
111     aes32esmi x19, x19, x20, 1
112     aes32esmi x19, x19, x21, 2
113     aes32esmi x19, x19, x22, 3          // x16-x19 contains new state
114
115     j .aes128_enc_blk_l0
116     xor x0,x0,x0; xor x0,x0,x0; xor x0,x0,x0;
117
118 .aes128_enc_blk_final:
119     aes32esi x16, x16, x20, 0         // Odd Round [Final]
120     aes32esi x16, x16, x21, 1
121     aes32esi x16, x16, x22, 2
122     aes32esi x16, x16, x23, 3
123     aes32esi x17, x17, x21, 0
124     aes32esi x17, x17, x22, 1
125     aes32esi x17, x17, x23, 2
126     aes32esi x17, x17, x20, 3
127     aes32esi x18, x18, x22, 0
128     aes32esi x18, x18, x23, 1
129     aes32esi x18, x18, x20, 2
130     aes32esi x18, x18, x21, 3
```

```
131     aes32esi x19, x19, x23, 0
132     aes32esi x19, x19, x20, 1
133     aes32esi x19, x19, x21, 2
134     aes32esi x19, x19, x22, 3 // x16-x19 contains new state
135
136     li  t0, 0x1E1             // D=max, bank=1
137     .aes128_enc_blk_unmask:
138         csrrw x0, 0x006, t0   // Select i'th bank
139         sw x16, 0*4(a0)       // Store results
140         sw x17, 1*4(a0)
141         sw x18, 2*4(a0)
142         sw x19, 3*4(a0)
143         addi a0, a0, 16       // increment ct pointer
144         addi t0, t0, 1
145         bltu t0, t1, .aes128_enc_blk_unmask
146
147     csrrw x0, 0x006, x0         // Turn off SME
148
149     lw      x18, 0*4(sp)        // Saved reg restore
150     lw      x19, 1*4(sp)
151     lw      x20, 2*4(sp)
152     lw      x21, 3*4(sp)
153     lw      x22, 4*4(sp)
154     lw      x23, 5*4(sp)
155     addi    sp, sp,32           // Stack restore
156     ret
157
158 .endfunc
```