Implementing the Draft RISC-V Scalar Cryptography Extensions

Ben Marshall
ben.marshall@bristol.ac.uk
University of Bristol, Department of Computer Science
Bristol, United Kingdom

Daniel Page
daniel.page@bristol.ac.uk
University of Bristol, Department of Computer Science
Bristol, United Kingdom

Thinh Pham
thinh.pham@bristol.ac.uk
University of Bristol, Department of Computer Science
Bristol, United Kingdom

ABSTRACT

RISC-V is an increasingly popular, free and open Instruction Set Architecture (ISA). Many standard extensions to RISC-V are currently being designed and evaluated, including one for accelerating cryptographic workloads. Unlike most incumbent ISAs which re-use existing large SIMD state and data-paths to accelerate cryptographic operations, RISC-V also adds support for smaller machines with narrow 32 and 64-bit data-paths. For embedded, IoT class devices, this significantly lowers the barrier to entry for secure and efficient accelerated cryptography. In this paper, we describe (to our knowledge) the first complete, free and open-source implementation of the draft 32-bit RISC-V Cryptography Extension. We detail the performance benefits for several important algorithms, and associated hardware costs. Our experiences help to guide the ongoing standardisation work and provide a platform for other researchers to experiment with a complete and representative CPU system, implementing the draft cryptography extension.

CCS CONCEPTS

• **Security and privacy → Security in hardware**; • **Computer systems organization → Embedded systems**.

KEYWORDS

RISC-V, cryptography, ISE, implementation, CPU

ACM Reference Format:
Ben Marshall, Daniel Page, and Thinh Pham. 2020. Implementing the Draft RISC-V Scalar Cryptography Extensions. In *Hardware and Architectural Support for Security and Privacy (HASP '20), October 17, 2020, Virtual, Greece.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3458903.3458904

1 INTRODUCTION

RISC-V is a (relatively) new Instruction Set Architecture (ISA), which by design is, free and open for anyone to implement and extend [17]. As a result of these features, coupled with the surrounding community and availability of supporting infrastructure such as compilation tool-chains, a wide range of academic and industrial RISC-V implementations exist.

In the context of low-end embedded or IoT-class devices, the security of data during computation and communication is of critical importance. The delivery of security-related properties will typically depend on some form of cryptographic technology. Such technologies now span a rich set of functionality (i.e., beyond encryption), and, in theory at least, can delivery robust guarantees about the secrecy and authenticity of data and end-points. However, the practical implementation of cryptographic technologies still remains a significant challenge due to 1) the constraints low-end devices operate under, which include latency, throughput, area, memory footprint, and energy consumption, for example, and 2) the diverse attack landscape low-end devices operate in.

To address such challenges, a broad range of implementation strategies are available. At one end of the spectrum are pure software solutions. The implied constraint on computation and storage often means such a solution is unsuitable. For example, implementing AES in software for a small 32-bit CPU may imply hundreds of cycles per block encryption; improving this is difficult without the memory footprint associated with pre-computation. At the other end of the spectrum are pure hardware solutions. For low-end devices, such solutions often involve including a dedicated accelerator in a host system: see, e.g., the OpenTitan AES hardware IP[1]. While efficient in terms of latency, disadvantages of this approach include their low flexibility: such accelerators are usually fixed function, making it difficult or impossible to use other operating modes, or incorporate a specific countermeasure against a pertinent implementation attack. Where such an accelerator is not tightly-coupled (i.e. it is a physically separate, memory-mapped device), two further disadvantages may exist. First, the fact the accelerator is unable to share logic with the core leads to high area. Even when measured relative to the wider System on Chip (SoC), anything other than full utilisation of the accelerator, which would be unlikely, multiplies the associated overhead. Second, if the overhead of communication with the accelerator cannot be amortised, e.g., when encrypting short messages, this may dominate overall latency.

Positioned between pure software and pure hardware, hybrid solutions such as Instruction Set Extensions (ISEs) offer a compromise. In a general sense, an ISE is a special-purpose extension of a more general-purpose base ISA: while adding functionality, they can often allow improvements wrt. area and utilisation vs. hardware only alternatives, and latency and memory footprint vs. software only alternatives. ISEs are a native concept in RISC-V, where a small base ISA of only ~ 50 instructions is extended to suit different use-cases via either standard or custom ISEs. Multiple proposals for standard extensions are currently being worked on, one of which is the cryptography extension (Crypto ISE). The current proposal (detailed

[1] https://docs.opentitan.org/hw/ip/aes/doc/

HASP '20, October 17, 2020, Virtual, Greece

ACM ISBN 978-1-4503-8898-6/20/10…$15.00
https://doi.org/10.1145/3458903.3458904

in Section 2) is radically different to past ISAs in that it adds dedicated support for accelerating cryptography to deeply embedded 32-bit CPUs, all the way to sever class machines with large vector compute engines. It does this by adopting a tailored approach to each class of CPU core, with separate specifications for small, *scalar* cores and larger cores which implement the work-in-progress *vector* extension [23].

Consider, for example, support for AES. First, note that a strongly RISC-oriented design ethos (such as that of RISC-V) constrains the ISE design by, e.g., limiting instructions to a 3-address (2 source, 1 destination) format. This would rule out the design used by SPARC [9, Sections 7.3+7.4] because it employs a 4-address (3 source, 1 destination) alternative. Second, designs such as x86 [10, Section 12.13] or IBM Power [11] and ARMv8-A [15] are often retrofitted onto the base ISE, (re)using existing architectural resources; common example is a SIMD or vector register file, which better matches the word size required. RISC-V cannot rely only on this approach, as it leaves low end devices out of scope, hence the tailored support for difference classes of CPU.

Contributions. We describe the SCARV CPU and SoC platform: A complete implementation of a 32-bit RISC-V system, built with free and open-source tooling and designed to be a solid base which other researchers can build upon. Using the SCARV CPU as a base, we then describe (to our knowledge) the first complete, free and open-source implementation of the 32-bit RISC-V Crypto ISE. We provide evaluations of important cryptographic workloads using the ISE and the associated implementation costs, with comparisons to similar efforts and alternative solutions to cryptographic acceleration. We also give a qualitative evaluation of the functional verification effort needed to show the implementation is correct. We hope to contribute to the standardisation process by providing an implementation of the ISE which is completely open for scrutiny.

Organisation. Section 2 gives an overview of the RISC-V Crypto ISE proposal, focusing on the scalar instructions. Section 3 describes our base CPU and SoC platform. Section 4 details the implementation of the Crypto ISE, and the challenges of integrating it into an existing CPU. Section 5 describes the software size and performance improvements seen by including the ISE, and the associated hardware overheads.

2 THE DRAFT RISC-V CRYPTO ISE

The RISC-V Crypto ISE is in the process of being standardised, with draft releases [22] and work in progress code and infrastructure [21] available publicly on Github. For RISC-V in particular, the standardisation process is important to prevent fragmentation of the ISA, and to give a common platform for software developers to target. Were it not for standardised approaches to cryptography acceleration, it is likely that vendors would implement many different and incompatible custom extensions.

In this work, we refer to `v0.7.0` of the draft specification[2]. A thorough description of the ISE may be found there. We provide the following overview for completeness, and a listing of the implemented instructions in Appendix A. The Crypto ISE is broken into three main components.

2.1 Vector Instructions

The *Vector* component builds on the proposed RISC-V Vector Extension [23]. This component follows the longstanding tradition of using pre-existing SIMD or vector registers to implement large parts, or the entirety of certain cryptographic algorithms. For example, there is an *All-Rounds* AES encryption instruction, which takes a key and input state, and performs an entire AES block encryption in one instruction.

This component of the Crypto ISE is aimed at large cores, and so is not discussed further in this work, which focuses on embedded class devices.

2.2 Scalar Instructions

The *Scalar* component is aimed at smaller, resource constrained devices which do not implement the Vector Extension, but which still benefit from accelerated cryptography. It defines new instructions for the 32 and 64-bit base ISAs. Importantly, all of the instructions in the scalar component meet the 2-source and 1-destination constraint for register accesses. The instructions may be split into two main categories: generic instructions which are useful for cryptography but not specific to it; and algorithm specific instructions, which are very efficient, but useful only for their particular algorithm.

Parts of the official RISC-V Scalar Cryptography draft proposals have been contributed from or informed by the XCrypto [27] project, which also explores accelerated Cryptography on RISC-V. We view the official RISC-V Crypto ISE evaluated here as a much improved, more focused and standardisation appropriate cousin of the much larger and more experimental, research orientated XCrypto.

2.2.1 Generic Instructions. The Crypto ISE contains several generic instructions which are useful for low level cryptographic operations. Rotation is essential for many important block ciphers and hash functions, particularly the SHA3 and CSHAKE functions [20] which underpin several of the NIST Post-Quantum Cryptography standardisation candidates. Bit permutation instructions are useful components of many block ciphers, particularly for implementations of generic SBoxes. They are also useful for endianness conversions. Logical operations (and-not, or-not, exclusive nor) are useful for implementing software-based masking countermeasures for certain classes of side-channel. Carry-less multiplication is also included, due to its importance in the Galois/Counter-Mode of operation [19].

Importantly, all of these generic instructions are also found in the draft Bitmanipulation extension. Indeed, ownership of these instructions from a standardisation perspective lies with the RISC-V Bitmanip Task Group. By sharing (or borrowing) instructions from another extension, much work is saved in terms of specification, implementation and verification. It also means that software developers writing cryptographic code can always rely on the same critical instructions, rather than having to target potentially many variations of the Bitmanip extension.

[2]Note that we have undertaken this work because the specification has stabilised, and is considered *feature complete*. The exact version number does not represent or imply how *ready* it is.

```
sha256sig0 rd, rs1:
    rd <- ROR32(rs1,7) ^ ROR32(rs1,18) ^ SHR32(rs1, 3)

sha256sum0 rd, rs1:
    rd <- ROR32(rs1,2) ^ ROR32(rs1,13) ^ ROR32(rs1,22)
```

Figure 1: Pseudocode for the `sha256sum0/sig0` instructions. The `sha256sum1/sig1` instructions are the same in form, but with different shift and rotation constants.

2.2.2 Specialist Instructions: AES. There are two types of dedicated AES instructions: for 32 and 64-bit data-paths. Based on an evaluation in [26], the proposed design for 32-bit systems is based on [30]. It uses a "T-tables in hardware" approach, and requires only a single SBox instantiation. This is often the most expensive part of AES to implement in hardware, making the design much more suitable for embedded systems. The proposal requires 16 dedicated AES instructions per block encrypt/decrypt round, plus 4 load-word instructions to move the round keys into the general purpose registers. Figure 3 shows the generic data-path for the AES instructions.

2.2.3 Specialist Instructions: SHA2. The lightweight SHA2 instructions implement the *Sigma* and *Sum* transformations, described in [18, Sections 4.1.2, 4.1.3]. Example pseudocode for the instructions is shown in Figure 1.

For SHA256 these functions are implemented directly in a 32-bit data-path. For SHA512 the functions operate on 64-bit elements, so the functions are split across multiple instructions. Each instruction sources two 32-bit registers (making a single 64-bit input) and outputs either the high or low 32 bits of the result.

2.2.4 Specialist Instructions: SM3 and SM4. The SM3 instructions implement the $P0$ and $P1$ transformations of the SM3 hash function [6], and are analogous to the SHA2 instructions shown in Figure 1 but with slightly different shift and rotation configurations.

The SM4 instructions also use a "hardware T-table" based approach to accelerating the SM4 block cipher [7], wrapping up the SBox and linear transformations in a single instruction. The similarity of the data-paths for the SM4 and AES instructions leads to some interesting implementation choices, which are explored in Section 4.

2.3 Entropy Source

The rationale for the Entropy Source (ES) component of the Crypto ISE is explored more completely in [31]. We provide a short summary here. The key principle of the ES instruction is to provide an *architectural interface* to a source of entropy which can be used for generating cryptographic secrets. While there is considerable guidance to implementers in the specification, the core of the interface is very simple, constituting a single instruction: `pollentropy`. When executed, the instruction writes a single word to a GPR. The word contains 16-bits of random data, and a 2-bit status code. The status code indicates if the random data is *valid* or not. Three codes indicate why the data might not be usable: *WAIT* indicates the ES is still working to generate another data sample, *BIST* indicates the ES is performing an internal self test and *DEAD* indicates a fatal problem with the ES, and that it cannot be used. The random data output of `pollentropy` *must* be cryptographically conditioned before being used. `pollentropy` may *only* be executed in Machine-Mode.

The exact nature of the ES and how it is implemented is up to the designer to decide. A common choice might be a ring-oscillator. In this work, we strictly deal with implementation of the `pollentropy` instruction within the CPU, and do not discuss implementation of the ES itself, referring instead to [31].

3 THE CPU PLATFORM

Here, we describe our base CPU and SoC platform for computer architecture and side-channel security work. The RTL design, verification environments and implementation flows are available under an MIT License on Github [3, 4].

3.1 The SCARV CPU

Architecture. The SCARV CPU implements the 32-bit RISC-V base ISA and the Multiply and Compressed Standard Extensions [13, Chapters 7, 16]. The Multiply extension contains instructions for integer multiplication, division and remainder. The Compressed extension provides 16-bit variants of commonly used 32-bit base ISA instructions. It aims at improving code size and density, which is particularly important for embedded applications with limited memory and storage space. It also supports vectored interrupts, and the Machine-Mode section of the Privileged ISA specification [14]. From an architectural perspective, this makes it representative of many embedded micro-controller and IoT class CPUs in terms of features.

Micro-architecture. The micro-architecture of the SCARV CPU is shown in Figure 2. It implements a 5-stage pipeline, split into Fetch, Decode, Execute, Memory Access and Write-back. Operands are read in the Decode stage, with forwarding paths from the Execute, Memory and Write-back stages. The core has a 3-cycle load-to-use hazard penalty. As a micro-controller, the core does not support virtual memory and does not implement caches or dynamic branch prediction.

Functional Verification. We have spent considerable effort on the functional verification of the SCARV CPU. We re-use and extend the open `riscv-formal` framework[3] to formally verify the correctness of each instruction implementation, and the consistency of register accesses. All of the formal verification uses the free and open-source Yosys, SymbiYosys and Boolector tools. This makes it easy for other researchers to check for functional correctness before building on the SCARV CPU, without needing access to proprietary and expensive EDA tooling.

3.2 The SCARV SoC

The SCARV SoC is designed to provide the bare minimum of functionality needed to implement a representative embedded class CPU sub-system, without needing lots of complex software support packages. It wraps the SCARV CPU with a 1Kb ROM, 64Kb of RAM, some GPIO pins and a UART modem. The RAM size is configurable, with the default 64Kb size chosen to support the Embench IoT suite of benchmarks [29].

[3] https://github.com/SymbioticEDA/riscv-formal/

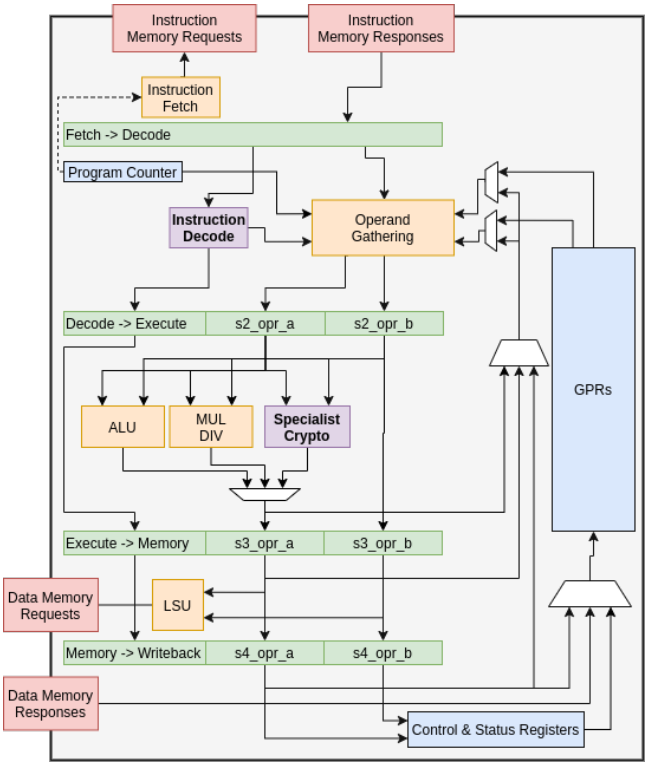

Figure 2: A block diagram of the SCARV CPU, showing the main execution pipeline and forwarding paths.

The SoC can be simulated using Verilator, a free and open Verilog simulator, or implemented on an FPGA. No FPGA or vendor specific code is used in the SoC (or CPU), but wrappers are provided to easily support fixed function blocks like BRAMs. There is a complete example project targeting the SASEBO-GIII [24], a side-channel analysis platform using a Xilinx Kintex-7 FPGA, with pre-generated bitfiles also available.

We plan to add support for the new OpenLANE ASIC implementation flow, an end-to-end synthesis, placement and routing flow built using free and open-source EDA tooling.

4 CRYPTO ISE IMPLEMENTATION

Here we describe our implementation of Scalar and Entropy Source components of the Crypto ISE, and its integration into the SCARV CPU.

First, we split the Crypto ISE in to "ALU-like" instructions, and specialist instruction. The ALU-like instructions were either generic and useful outside the Crypto ISE, or which could be implemented more efficiently when considered alongside existing instructions. For example, the rotation instructions can be much more efficiently implemented when integrated into the existing shift instruction implementation.

We would also expect that the more generic instructions would be used more frequently than the specialised instructions, which are used very intensely in short bursts. By separating the specialised instructions into their own functional unit, we can gate the inputs to those instructions to prevent unnecessary toggling. This is essential in energy efficient designs.

Table 1: Segmenting the ALU-like and specialist instructions.

ALU-like instructions	Specialist Instructions
`grev/gorc`, `shfl`, `clmul`, `pack`, `andn`, `orn` `xorn`, `xperm`, Rotations	SHA2-256, SHA2-512, AES, SM3, SM4

Our final segmentation is shown in Table 1, with ALU-like instructions implemented as part of the existing integer ALU, and specialist instructions grouped together in their own functional unit.

4.1 ALU-like Instruction Implementation

The `andn`, `orn` and `xorn` instructions can be implemented very easily with the base ISA `and`, `or` and `xor` instructions by inverting the second operand as needed.

The rotation instructions (`rol`, `ror` and `rori`) may share most of their logic with the base ISA shift instructions. Our implementation uses a barrel shifter, where either: ones, zeros or a repetition of the shifted operand is shifted in, based on whether we are doing an arithmetic, logical or rotary shift.

The generic reverse instructions (`grev`, `grevi`, `gorc`) and permutation instructions (`xperm`, `shfl`) are implemented as dedicated logic, since they share none of their functionality with existing instructions.

The carry-less multiply instruction was implemented as part of the existing Multiplier, re-using the micro-architectural state therein. It is essential that the carry-less multiply (and indeed, integer multiply) instructions are implemented in a constant time manner. Any early-out mechanisms which introduce data-dependant execution times introduce a timing side channel, which can be exploited remotely.

4.2 Specialist Instruction Implementation

For the specialist instructions, we again segment into two groups: the hash function instructions (SHA2, SM3) and the block cipher instructions (AES, SM4).

The SHA2 and SM3 instructions are very similar in their design. They both consist of operands which are rotated or shifted by several constants. The rotated/shifted versions of the operands are then xor'd together. The instructions are very lightweight, but with little shared logic between them.

The AES and SM4 instructions share similar high-level data-paths, as shown in Figure 3. A tradeoff between size and circuit depth can hence be made by combining (or not) the two instructions into a single shared data-path. We examine the results of this tradeoff in Section 5. The area of these instructions is dominated by their SBox Implementations.

4.3 `pollentropy` Implementation

We implement the ES as a memory mapped peripheral. The `pollentropy` instruction is then translated into a base ISA *load word* instruction,

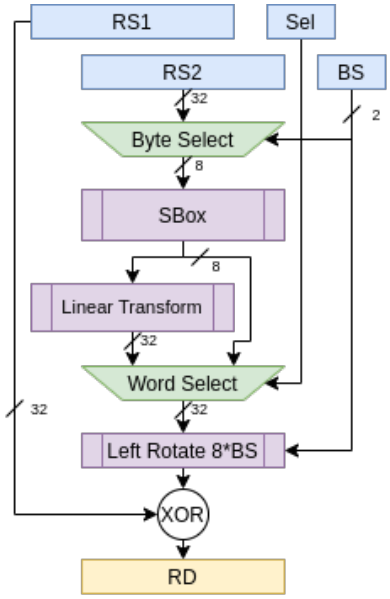

Figure 3: Data-paths of the AES and SM4 instructions. The SBox and Linear Transformation blocks differ between the instructions, but all other data-path aspects remain the same.

to a pre-determined memory address. This greatly simplifies the integration, and allows for clean separation between the CPU and the ES, which is important for verification purposes.

Additionally, it allows for *vendor specific* access to the raw output of the entropy source to be added via additional memory mapped registers. Access to this raw output is required for various validation programs such as FIPS 140-3 [12] and AIS-31 [25]. After post-manufacturing validation, such raw data access should be disabled to prevent any possible security vulnerabilities.

This means that the additional logic for translating the `pollentropy` instruction into a *load word* is included in the subsequent evaluation section, but the actual ES is not. We consider this to be a totally separate component, worthy of dedicated analysis, as found in [31].

5 EVALUATION

Here we evaluate the implementation cost of the Crypto ISE, using the SCARV CPU as a base. We first look at the instruction functional units in isolation, followed by the integration into the wider system. We then evaluate the performance and code-size impacts of several important cryptographic workloads, and briefly discuss the impacts on verification infrastructure and effort.

To measure the implementation impact, we use Yosys (0.9-1706) [34] to synthesise the circuit, targeting a generic CMOS logic library. From this, we extract two metrics: the area of the synthesised circuit in NAND2 equivalent cells, and the topological depth of the circuit. We deliberately provide a more abstract evaluation, rather than specific timing/frequency and area measurements which are tied to a specific process design kit (PDK). This makes it easier for researchers and professionals to replicate and compare our results, without needing access to (often proprietary) PDKs, or expensive EDA tooling. It also matches the methodology used to evaluate the Bit-manipulation extension [2, Section 3.1], making our results directly comparable.

Table 2: Comparison of hardware module sizes when synthesised for a generic CMOS cell library.

	Baseline RV32IMC		+Crypto ISE	
Module	Size	Depth	Size	Depth
ALU	1362	34	4215	34
Multiplier	5634	47	5912	47
Specialist Instructions				
→ SHA256		N/A	737	5
→ SHA512		N/A	845	6
→ SM3		N/A	474	3
→ SM4		N/A	693	29
→ AES		N/A	1174	31
→ AES + SM4		N/A	1429	36
CPU, Separate AES+SM4	33386	67	39929	69
CPU, Combined AES+SM4	"	"	39250	69

Table 3: Comparison of hardware modules when implemented on a Xilinx Kintex-7 FPGA, for a 50MHz target frequency.

Design	Slices	LUTs	FFs	Timing Slack
Baseline RV32IMC	1221	3843	1808	10.302 ns
+ISE Separate AES+SM4	1817	5834	1813	8.929 ns
+ISE Combined AES+SM4	1787	5713	1813	8.913 ns

Table 4: Sizes of commercial AES and SHA2 accelerators, targeting Xilinx Kintex-7 FPGAs and $0.13\mu m$ CMOS ASIC.

Accelerator	ASIC Gates	FPGA Slices	Throughput ASIC/FPGA	Source
AES-128 Enc + Dec	<5K	94	100/270Mbps	[1]
SHA1 + SHA-256	<9K	262	100/182Mbps	[5]

5.1 Hardware Overheads

Each modified functional unit was first evaluated in isolation from the main CPU. This allows a clearer picture of the instruction implementation costs compared to the base ISA instructions, without the costs being hidden by the size of the host CPU system. The results are shown in Table 2.

The ALU has increased in size by 3×. This is unsurprising given how small the base RISC-V ISA is. Notably, there is no effect on the circuit depth. The Multiplier has grown very modestly, reflecting how the carry-less addition at the core of carry-less multiply is very cheap (an `xor`). This is despite our `clmul` implementation using an unrolled strategy, and taking only 5 cycles to produce the full result. Note that the integer multiplier also uses an unrolled implementation strategy, though is less aggressive; taking 9 cycles

to complete. In both cases, the unrolling depth is parameterised, making tuning for area or latency very easy. We have not implemented macro-op fusion[4] to allow a `clmulh` and `clmul` to be fused into one instruction, though this is an obvious future optimisation.

For the Specialist Crypto ISE instructions, the additional area is dominated by the AES and SM4 instruction implementations. Implementing the AES and SM4 instructions using shared datapaths is $0.76x$ the size, compared to implementing them separately, at the cost of a larger circuit depth. The standalone AES module is much larger than the SM4 module due to AES needing separate SBoxes for encryption and decryption, while SM4 uses only a single SBox for both encryption and decryption.

Overall, the area overhead for the CPU subsystem when targeting a generic CMOS library is $1.20x$ and $1.18x$ for the separate and combined AES/SM4 implementations respectively.

We also implemented the SCARV CPU on a Xilinx Kintex-7 FPGA (`xc7k160tfbg676-3`) using Vivado 2019.2 with default synthesis options. The results are shown in Table 3. All systems target the same 50MHz operating frequency, with the available Timing Slack showing the impact on critical path length. Slice overheads are $1.49x$ and $1.46x$ for separate and combined AES/SM4 implementations respectively. We expect the larger overheads on the FPGA are due to the relative inflexibility of FPGA LUT primitives compared to gate level circuit primitives in the CMOS library. For example, a 2-input function on the FPGA may still be implemented using a 4-input LUT, artificially increasing resource usage. Likewise, logic slices may not be fully utilised, particularly in designs with lots of un-structured logic like the AES and SM4 SBoxes.

Direct comparisons to past work are difficult, as it is rare to find open-source evaluations of entire ISEs. We follow the example of the Draft RISC-V Bitmanipulation extension, and compare the area overhead of the Crypto ISE dedicated logic to the Rocket Core [16] (The Rocket Core is another popular open source implementation of RISC-V) Multiply/Divide unit (MDU), and a basic implementation of the Bitmanipulation extension. The results are listed in Table 5. The entire Crypto ISE is slightly larger than the Rocket Core Multiply Divide unit. We note the similarity in size between the SCARV CPU and Rocket MDUs. We note that the Crypto ISE contains instructions which are also implemented in the Bitmanipulation `ZBB` profile [2, Chapter 2, Page 3], namely the logic-and-invert instructions, and the rotation instructions.

When comparing to commercial, fixed function cryptographic accelerators, the Crypto ISE is an interesting compromise between performance and area. Observing Table 4, we see that the CMOS area overhead of the entire Crypto ISE is similar to the cost of a single area-optimised AES-128 accelerator, and is considerably smaller than a SHA accelerator. For embedded SoCs, which do not have die space for dedicated cryptographic accelerators, we believe that these results make a compelling case for the RISC-V Crypto ISE as a lightweight and more flexible alternative.

5.2 Software

Table 6 and Table 7 shows the results of running several cryptographic workloads targeting the baseline ISA, and the Crypto ISE.

[4] Macro-op fusion is a technique for combining multiple ISA instructions into a single micro-op executed by the CPU, resulting in a performance advantage. RISC-V proposes to take advantage of this technique in many places, but it can be complex to do in small, embedded CPUs with narrow fetch buffers.

Table 5: Comparison of overheads between the Crypto ISE, Draft RISC-V Bitmanip extension and the Rocket Core Multiply Divide Unit. Note that the component overheads of the Crypto ISE do not sum to the total overhead figure. This is a result of synthesising components in isolation, where cross-module optimisations cannot be applied.

Module	Size (NAND2 Gates)
Bitmanip (ZBB RV32)	2471
Rocket RV32 MulDiv	5167
Crypto ISE Total	5864
→ Crypto ISE ALU Overhead	2853
→ Crypto ISE MDU Overhead	278
→ Crypto ISE Specialist	3485

Table 6: Comparison of software performance changes with the base and extended instruction sets.

	Baseline RV32IMC		+Crypto ISE	
Algorithm	Instrs	Cycles	Instrs	Cycles
AES128 Enc	1024	1665	241	296
SM4 Enc	1434	2025	290	496
SHA2-256	3758	4808	1682	2143
SM3	3858	4961	2170	2445
GHASH	3486	3919	111	376
ChaCha20	1714	2000	1068	1275

We include: block encryption functions for AES-128 [8] and SM4 [7]; the hashing of a single message block for SHA256 [18]; SM3 [6] and GHASH [19]; and a single block permutation of the ChaCha20 stream cipher.

Clearly, the ISE drastically increases the software performance of the considered algorithms. The GHASH operation receives a disproportionate benefit from the carry-less multiply instructions. When implemented in software, carry-less multiply is extremely slow, particularly when the full 64-bit result is needed from multiplying two 32-bit operands, as is the case here.

Comparing to past work optimising AES on ARM [32, Table 1, Section 3.1], we can see the proposed lightweight RISC-V AES ISE is 2.23× faster, with zero data memory accesses or stack usage, likely resulting in a considerable energy saving. Likewise comparing to past work on optimised AES for RISC-V [33, Table 1, Section 7.1], we see the ISE is 3× faster than a heavily optimised T-Tables implementation on a `RV32IMC` micro-controller with caches.

We note that the Crypto ISE does not include any instructions aimed at accelerating Public Key Cryptography. This is already well supported by the base RISC-V ISA, and would not be affected by the Crypto ISE, hence we exclude it from our analysis. Although Public Key Cryptography is essential for embedded class cores to establish communications or for secure boot, the volume and frequency of Public Key operations is much smaller than Symmetric Key operations for most use cases. E.g. the volume of symmetrically encrypted data in a TLS session is much larger than that which is

Table 7: Comparison of code size changes with the base and extended instruction sets.

	Baseline RV32IMC		+Crypto ISE	
Algorithm	.text	.data	.text	.data
AES128 Enc	1014	4096	286	0
SM4 Enc	912	256	156	0
SHA2-256	4794	256	2368	256
SM3	6648	0	3832	0
GHASH	482	0	256	0
ChaCha20	694	0	464	0

used for key agreement or handshaking. Hence, we believe this is a reasonable design choice for the proposed ISE.

5.3 Verification

Any ISE is useless if its functional correctness cannot be verified in a timely manner; hence the verification effort needed to integrate the RISC-V Crypto ISE into an existing core is a useful consideration in any standardisation process.

We found that verifying the Crypto ISE integration was not disproportionate to its size or the complexity of any instruction specifications. Because we primarily use model-checking to verify the SCARV CPU, we extended the existing `riscv-formal` trace interface to add checkers for the new instructions. No new signals needed to be traced out of the core, since all instructions adhere to the 2-reads-1-write register access constraint. This made extending our verification environment considerably easier.

Three classes of instruction caused some difficulties for the formal verification environment: the carry-less multiply instructions, the non-linear parts of the AES and SM4 instructions and the entropy source instruction.

For the carry-less multiply and non-linear operations, the nature of these operations is such that formal tools (specifically, bounded model checking tools) struggle to handle them, which is a known shortcoming. We worked around this by creating small directed tests for known corner case inputs to these instructions, and forcing the formal tools to timeout after a set interval to cover as much of the state space as possible, and to check interactions with other instructions. Such problems would not appear in a simulation based verification strategy.

For the `pollentropy` instruction, the difficulty comes from the inherently random write-back value. Functionally verifying the instruction *does not* involve checking that quality of the randomness, only that the instruction executes and forwards its results correctly. For formal verification flows such as ours, we believe it is reasonable to leave the register write-back value un-specified, but to check that *whatever* value it does return is written back and forwarded to subsequent instructions correctly. Our decision to implement the `pollentropy` instruction as a *load word* made the verification much easier, as memory response values are at the boundary of our verification interface. We simply had to check the correct memory address was issued, and that privileged mode restrictions on executing `pollentropy` were met.

We anticipate no difficulties integrating any of the Crypto ISE instructions into a constrained random verification flow, such as a UVM [28] base environment. This is fairly self evident from how the instructions do not introduce new state, do not interact with the privileged architecture and all meet the 2-read-1-write register file access constraint. In this sense, they are similar to the base ISA instructions from the perspective of stimulus generation and coverage closure.

In the case of the `pollentropy` instruction, any golden reference model would need to have the write-back value of the instruction hinted too it, which is a standard way of dealing with non-deterministic behaviour in the functional verification of CPUs.

6 CONCLUSION & FUTURE WORK

We have implemented the Draft RISC-V Cryptography Extension for Scalar 32-bit CPUs. We find that it delivers excellent performance benefits to several important cryptographic workloads, with modest area and latency overhead to the host core. As the only architecture proposing dedicated cryptographic instructions for very small cores, RISC-V is at a considerable advantage to legacy architectures as a platform for secure IoT devices. Our free and open implementation of the ISE proposals will help guide the standardisation process, and give researchers a platform on which to conduct further research using the RISC-V cryptographic ISE.

Future work in this area might include further optimising the implementation, evaluating larger and more performance orientated designs, or hardening algorithms using the ISE against side-channel attack.

ACKNOWLEDGMENTS

We would like to thank the members of the RISC-V Cryptographic Extensions Task Group for welcoming us into the RISC-V standardisation process.

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

REFERENCES

[1] [n.d.]. AES core - Xilinx, Altera, Microsemi, Lattice and ASIC - Helion Technology. https://www.heliontech.com/aes_tiny.htm. Retrieved Sept 7th, 2020.
[2] [n.d.]. RISC-V Bit manipulation extension draft proposal. https://github.com/riscv/riscv-bitmanip/blob/master/bitmanip-draft.pdf
[3] [n.d.]. SCARV CPU - and open source 32-bit RISC-V CPU for research. https://github.com/scarv/scarv-cpu. RISC-V Cryptography ISE Development Branch.
[4] [n.d.]. SCARV SoC - and open source RISC-V base SoC for research. https://github.com/scarv/scarv-soc. RISC-V Cryptography ISE Development Branch.
[5] [n.d.]. SHA-1, SHA-2, MD5 Tiny Hashing Cores for FPGA (Xilinx, Altera) - Helion Technology. https://www.heliontech.com/tiny_hash.htm. Retrieved Sept 7th, 2020.
[6] [n.d.]. The SM3 Cryptographic Hash Function. https://tools.ietf.org/id/draft-oscca-cfrg-sm3-02.html. Retrieved 12th March, 2020.
[7] [n.d.]. The SM4 Block Cipher Algorithm And Its Modes Of Operations. https://tools.ietf.org/id/draft-crypto-sm4-00.html. Retrieved 26th March, 2020.
[8] 2001. Advanced Encryption Standard (AES). National Institute of Standards and Technology (NIST) Federal Information Processing Standard (FIPS) 197. https://www.nist.gov/publications/advanced-encryption-standard-aes
[9] 2016. *Oracle SPARC Architecture 2011.* Technical Report D1.0.0. Oracle Corp. https://www.oracle.com/technetwork/server-storage/sun-sparc-enterprise/documentation/140521-ua2011-d096-p-ext-2306580.pdf.
[10] 2018. *Intel 64 and IA-32 architectures – Software Developer's Manual (Volume 1: Basic Architecture).* Technical Report 325383-067US. Intel Corp. http://software.intel.com/en-us/articles/intel-sdm.
[11] 2018. *Power ISA.* Technical Report 2.07 B. IBM. https://ibm.ent.box.com/s/jd5w15gz301s5b5dt375mshpq9c3lh4u.
[12] 2019. FIPS 140-3: Security Requirements For Cryptographic Modules. https://doi.org/10.6028/NIST.FIPS.140-3
[13] 2019. *The RISC-V Instruction Set Manual.* Technical Report Volume I: User-Level ISA (Version 20190608-Base-Ratified). http://riscv.org/specifications/

[14] 2019. *The RISC-V Instruction Set Manual.* Technical Report Volume II: Privileged Architecture (Version 20190608-Priv-MSU-Ratified). http://riscv.org/specifications/
[15] ARM 2020. *Arm Architecture Reference Manual: Armv8, for Armv8-A architecture profile* (DDI0487F.a ed.). ARM. https://static.docs.arm.com/ddi0487/fa/DDI0487F_a_armv8_arm.pdf.
[16] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. 2016. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* (2016).
[17] K. Asanović and D.A. Patterson. 2014. *Instruction Sets Should Be Free: The Case For RISC-V.* Technical Report UCB/EECS-2014-146. http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html.
[18] Quynh H. Dang. 2015. *Secure Hash Standard.* Number NIST FIPS 180-4. https://doi.org/10.6028/NIST.FIPS.180-4
[19] Morris J Dworkin. 2007. *Sp 800-38d. recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac.* National Institute of Standards & Technology.
[20] Morris J Dworkin. 2015. *SHA-3 standard: Permutation-based hash and extendable-output functions.* Technical Report. National Institute of Standards and Technology. https://doi.org/10.6028/NIST.FIPS.202
[21] RISC-V Cryptography Task Group. [n.d.]. RISC-V Cryptographic Extension Proposals Github Repository. https://github.com/riscv/riscv-crypto.
[22] RISC-V Cryptography Task Group. [n.d.]. *RISC-V Scalar Cryptographic Extension Draft Proposal v0.7.0.* Technical Report. https://github.com/riscv/riscv-crypto/releases/v0.7.0.
[23] RISC-V Vector Extension Task Group. [n.d.]. RISC-V Vector Extension Github Repository. https://github.com/riscv/riscv-v-spec.
[24] Y. Hori, T. Katashita, A. Sasaki, and A. Satoh. 2012. SASEBO-GIII: A hardware security evaluation board equipped with a 28-nm FPGA. In *IEEE Global Conference on Consumer Electronics.* 657–660.
[25] Wolfgang Killmann and Werner Schindler. 2011. AIS-31 test suites. A Proposal for: Functionality classes for random number generators.
[26] Ben Marshall, G. Richard Newell, Dan Page, Markku-Juhani O. Saarinen, and Claire Wolf. [n.d.]. The design of scalar AES Instruction Set Extensions for RISC-V. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021, 1 ([n. d.]). To appear.
[27] Ben Marshall, Daniel Page, and Thinh Pham. 2019. *XCrypto: a cryptographic ISE for RISC-V.* Technical Report. Tech. rep. 1.0. 0. 2019.
[28] Ashok B Mehta. 2018. UVM (Universal Verification Methodology). In *ASIC/SoC Functional Design Verification.* Springer, 17–64.
[29] David Patterson, Jeremy Bennett, Palmer Dabbelt, Cesare Garlati, G. S. Madhusudan, and Trevor Mudge. [n.d.]. Embench: Open Benchmarks for Embedded Platforms. https://github.com/embench/embench-iot.
[30] Markku-Juhani O. Saarinen. 2020. A Lightweight ISA Extension for AES and SM4. In *First International Workshop on Secure RISC-V Architecture Design Exploration (SECRISC-V'20).* IEEE. https://arxiv.org/abs/2002.07041
[31] Markku-Juhani O. Saarinen, G. Richard Newell, and Ben Marshall. 2020. Building a Modern TRNG: An Entropy Source Interface for RISC-V. In *4th Workshop on Attacks and Solutions in Hardware Security (ASHES'20), November 13, 2020, Virtual Event, USA.* ACM. https://doi.org/10.1145/3411504.3421212
[32] Peter Schwabe and Ko Stoffelen. 2017. All the AES You Need on Cortex-M3 and M4. In *Selected Areas in Cryptography – SAC 2016*, Roberto Avanzi and Howard Heys (Eds.). Springer International Publishing, Cham, 180–194.
[33] Ko Stoffelen. 2019. Efficient Cryptography on the RISC-V Architecture. In *Progress in Cryptology – LATINCRYPT 2019*, Peter Schwabe and Nicolas Thériault (Eds.). Springer International Publishing, Cham, 323–340.
[34] Claire Wolf. [n.d.]. Yosys Open SYnthesis Suite. http://www.clifford.at/yosys/.

A INSTRUCTION SET LISTING

Here, we give a very brief listing of all instructions implemented as part of the 32-bit RISC-V Scalar Cryptography ISE, `v0.7.0`. Their complete functional descriptions can be found in [22]. We would have included complete semantics for each instruction if they could fit, but prefer to point to the official draft specification.

```
# 32-bit Rotations
ror             rd, rs1, rs2
rol             rd, rs1, rs2
rori            rd, rs1, imm

# Permutation Instructions
grev            rd, rs1, rs2
grevi           rd, rs1, imm
gorc            rd, rs1, rs2
shfl            rd, rs1, rs2
unshfl          rd, rs1, rs2
shfli           rd, rs1, imm
unshfli         rd, rs1, imm
xperm.n         rd, rs1, rs2
xperm.b         rd, rs1, rs2

# Carry-less Multiply
clmul           rd, rs1, rs2
clmulh          rd, rs1, rs2
clmulr          rd, rs1, rs2

# Logic-and-Negate
andn            rd, rs1, rs2
orn             rd, rs1, rs2
xorn            rd, rs1, rs2

# Byte/halfword packing
pack            rd, rs1, rs2
packu           rd, rs1, rs2
packh           rd, rs1, rs2

# Scalar 32-bit AES instructions
aes32es         rd, rs1, rs2
aes32esm        rd, rs1, rs2
aes32ds         rd, rs1, rs2
aes32dsm        rd, rs1, rs2

# Scalar 32-bit SHA-256 instructions
sha256sum0      rd, rs1
sha256sum1      rd, rs1
sha256sig0      rd, rs1
sha256sig1      rd, rs1

# Scalar 32-bit hi/lo SHA-512 instructions
sha512sum0r     rd, rs1, rs2
sha512sum1r     rd, rs1, rs2
sha512sig0l     rd, rs1, rs2
sha512sig0h     rd, rs1, rs2
sha512sig1l     rd, rs1, rs2
sha512sig1h     rd, rs1, rs2


# Scalar 32-bit SM3 instructions.
sm3p0           rd, rs1
sm3p1           rd, rs1

# Scalar 32-bit SM4 instructions.
sm4ed           rd, rs1, rs2
sm4ks           rd, rs1, rs2

# Entropy Source Instruction.
pollentropy     rd, imm
```