A Lightweight ISA Extension for AES and SM4

Markku-Juhani O. Saarinen

PQShield Ltd.
Oxford, United Kingdom
mjos@pqshield.com

***Abstract*—We describe a lightweight RISC-V ISA extension for AES and SM4 block ciphers. Sixteen instructions (and a subkey load) is required to implement an AES round with the extension, instead of 80 without. An SM4 step (quarter-round) has 6.5 arithmetic instructions, a similar reduction. Perhaps even more importantly the ISA extension helps to eliminate slow, secret-dependent table lookups and to protect against cache timing side-channel attacks. Having only one S-box, the extension has a minimal hardware size and is well suited for ultra-low power applications. AES and SM4 implementations using the ISA extension also have a much-reduced software footprint. The AES and SM4 instances can share the same data paths but are independent in the sense that a chip designer can implement SM4 without AES and vice versa. Full AES and SM4 assembler listings, HDL source code for instruction's combinatorial logic, and C code for emulation is provided to the community under a permissive open source license. The implementation contains depth- and size-optimized joint AES and SM4 S-Box logic based on the Boyar-Peralta construction with a shared non-linear middle layer, demonstrating additional avenues for logic optimization. The instruction logic has been experimentally integrated into the single-cycle execution path of the "Pluto" RV32 core and has been tested on an FPGA system.**

***Index Terms*—RISC-V, AES, SM4, Cryptographic ISA Extension, Lightweight Cryptography**

I. Introduction

The Advanced Encryption Standard (AES) is a 128-bit block cipher with 128/192/256 - bit key, defined in the FIPS 197 standard [9]. AES is a mandatory building block of the TLS 1.3 [12] security protocol and is widely used for storage encryption, shared-secret authentication, cryptographic random number generation, and in many other applications.

The SM4 block cipher [13] fulfills a similar role to AES in the Chinese market and is the main block cipher recommended for use in China. It is also standardized with ISO [7]. SM4 also has a 128-bit block size, but only one key size, 128 bits. Even though its high-level structure differs completely from AES, the two share significant similarities in their sole nonlinear component, which is a single 8×8-bit "S-Box" substitution table in both cases.

Cache timing attacks on AES became well known in mid-2000s when it was demonstrated that common table-based implementations can be exploited even remotely [2], [11]; very similar issues also affect SM4. In presence of a cache, the only way to make the execution time of these ciphers fully independent of secret data is to eliminate the table lookup either by implementing it as bitsliced Boolean logic or by providing a specific ISA extension for the S-Box lookup.

Consumer CPUs have had instructions to support AES for almost a decade via the Intel AES-NI in x86 [6] and ARMv8-A cryptographic extensions [1]; these are almost universally available in PCs and higher-end mobile devices such as phones. ARM also supports SM4 via the ARMv8.2-SM extension. The AES instructions have been shown to make AES much less of a throughput bottleneck for high-speed TLS communication (servers) and storage encryption (mobile devices), thereby also extending battery life in the latter. Both Intel and ARM cryptographic ISAs require 128-bit (SIMD) registers, and are not available on lower-end CPUs.

In this work, we show that it is possible to create a simple AES and SM4 ISA extension that offers a significant performance improvement and timing side-channel resistance with a minimally increased hardware footprint. It is especially suitable for lightweight RV32 targets.

II. A Lightweight AES and SM4 ISA Extension

The ISA extension operates on the main register file only, using two source registers, one destination register, and a 5-bit field `fn[4:0]` which can be seen either as an "immediate constant" or just code points in instruction encoding. In either case, the interface to the (reference) combinatorial logic is:

```
module saes32(
  output [31:0] rd,    // to output register
  input  [31:0] rs1,   // input register 1
  input  [31:0] rs2,   // input register 2
  input  [4:0]  fn     // 5-bit func specifier
);
```

See Section IV-B for encoding details of SAES32 as an RV32 R-type custom instruction for testing purposes. For RV64 the words are simply truncated or zero-extended.

The five bits of `fn` cover encryption, decryption, and key schedule for both algorithms. Bits `fn[1:0]` first select a single byte from `rs2`. Two bits `fn[4:3]` indicate which $8 \rightarrow 8$ - bit S-Box is used (AES, AES^{-1}, or SM4), and additionally `fn[4:2]` specifies a $8 \rightarrow 32$ - bit linear expansion transformation (each of three S-Boxes has two alternative linear transforms, indicated by `fn[2]`). The expanded 32-bit

This work was supported by Innovate UK (R&D Project Ref. 105747.)

First International Workshop on Secure RISC-V Architecture Design Exploration (SECRISC-V'20). It is held in conjunction with the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) - August 23rd, 2020 in Boston, Massachusetts, USA.

TABLE I
HIGH-LEVEL ASSEMBLER IDENTIFIERS FOR FN[4:2].

Instruction	fn[4:2]	Description or Use
`saes32.encsm`	`3'b000`	AES Encrypt round.
`saes32.encs`	`3'b001`	AES Final / Key sched.
`saes32.decsm`	`3'b010`	AES Decrypt round.
`saes32.decs`	`3'b011`	AES Decrypt final.
`ssm4.ed`	`3'b100`	SM4 Encrypt and Decrypt.
`ssm4.ks`	`3'b101`	SM4 Key Schedule.
Unused	`3'b11x`	($4 \times 6 = 24$ points used.)

value is then rotated by 0–3 byte positions based on `fn[1:0]`. The result is finally XORed with `rs1` and written to `rd`.

Table I contains the identifiers (pseudo instructions) that we currently use for bits `fn[4:2]`. Usually we may arrange computation so that `rd` = `rs1` without increasing instruction count, making a two-operand "compressed" encoding possible.

For AES the instruction selects a byte from `rs2`, performs a single S-box lookup (*SubBytes* or its inverse), evaluates a part of the MDS matrix (*MixColumns*) if that linear expansion is step selected, rotates the result by a multiple of 8 bits (*ShiftRows*), and XORs the result with `rs1` (*AddRoundKey*). There is no need for separate instructions for individual steps of AES as small parts of each of them have been incorporated into a single instruction. We've found that each one of these substeps requires surprisingly little additional logic.

For SM4 the instruction has the same data path with byte selection, S-Box lookup, and two different linear operations, depending on whether encryption/decryption or key scheduling task is being performed.

Both AES [9] and SM4 [13] specifications are written using big-endian notation while RISC-V uses primarily little-endian convention [15]. To avoid endianness conversion the linear expansion step outputs have a flipped byte order. This is less noticeable with AES, but the 32-bit word rotations of SM4 become less intuitive to describe (while wiring is equivalent).

We refer to the concise reference implementation discussed in Section IV for details about specific logic operations required to implement the ISA extension, and for standards-derived unit tests.

III. USING THE AES AND SM4 INSTRUCTIONS

AES and SM4 were originally designed primarily for 32-bit software implementation. SAES32/SSM4 adopts the "intended" 32-bit implementation structure but removes table lookups and rolls several individual steps into the same instruction. Both AES and SM4 implementations are also realizable with the reduced "E" register file without major changes.

A. AES Computation and Key Schedule

The structure of an AES implementation is similar to a "T-Table" implementation, with sixteen invocations of `saes32.encsm` per round and not much else (apart from fetching the round subkeys). In practice, two sets of four registers are used to store the state, with one set being used to rewrite the other, depending on whether an odd or even-numbered round is being processed. AES has $r \in \{10, 12, 14\}$ rounds, depending on the key size which can be $\{128, 192, 256\}$, respectively. The final round requires sixteen invocations of `saes32.encs`. The same instructions are also used in the key schedule which expands the secret key to $4r+4$ subkey words.

The inverse AES operation is structured similarly, with 16 `saes32.decsm` per main body round and 16 `saes32.decs` for the final round. These instructions are also used for reversing the key schedule. Four precomputed subkey words must be fetched in each round, requiring four loads (lw instructions) in addition to their address increment (typically every other round). There is no need for separate *AddRoundKey* XORs as the subkeys simply initialize either one of the four-register sets used to store the state.

It is also possible to compute the round keys "on the fly" without committing them to RAM. This may be helpful in certain types of security applications. The overhead is roughly 30%. However, if the load operation is much slower than register-to-register arithmetic, the overhead of on-the-fly subkey computation can become negligible. On-the-fly keying is more challenging in reverse.

B. SM4 Computation and Key Schedule

SM4 has only one key size, 128 bits. The algorithm has 32 steps, each using a single 32-bit subkey word. The steps are typically organized into 8 full rounds of 4 steps each. Due to its Feistel-like structure SM4 does not require an inverse S-Box for decryption like AES, which is a substitution-permutation network (SPN). The inverse SM4 cipher is equivalent to the forward cipher, but with a reversed subkey order.

Each step uses all four state words and one subkey word as inputs, replacing a single state word. Since input mixing is built from XORs, some of the temporary XOR values are unchanged and can be shared between steps. Each round requires ten XORs in addition to sixteen `ssm4.ed` invocations, bringing the total number of arithmetic instructions to 26 per round – or 6.5 per step. Therefore SM4 performance is slightly lower than that of AES-128, despite having fewer full rounds.

The key schedule similarly requires 16 invocations of `ssm4.ks` and 10 XORs to produce a block of four subkey words. The key schedule uses 32 "CK" round constants which can be either fetched from a table or computed with 8-bit addition operations on the fly.

For SM4 each block of four consecutive invocations of `ssm4.ed` and `ssm4.ks` share the same source and destination registers, differing only in `fn[1:0]` which steps through $\{0, 1, 2, 3\}$. We denote such a four-SAES32 block as pseudo instruction `ssm4.ed4`. One can reduce the per-round instruction count of SM4 from 26 (+4 lw) to 14 (+4 lw) by implementing `ssm4.ed4` as a "real" instruction that is almost four times larger than SAES32 in hardware.

Note that without additional instructions an AES implementation does *not* benefit from four parallel S-Boxes in encryption or decryption, only in key schedule.

TABLE II
ALGEBRAIC GATE COUNTS FOR A BOYAR-PERALTA TYPE LOW-DEPTH S-BOXES THAT IMPLEMENT SM4 IN ADDITION TO AES AND AES^{-1}.

Component	In, Out	XOR	XNOR	AND	**Total**
Shared middle	21 → 18	30	-	34	64
AES top	8 → 21	26	-	-	26
AES bottom	18 → 8	34	4	-	38
AES^{-1} top	8 → 21	16	10	-	26
AES^{-1} bottom	18 → 8	37	-	-	37
SM4 top	8 → 21	18	9	-	27
SM4 bottom	18 → 8	33	5	-	38

IV. REFERENCE IMPLEMENTATION

An open-source reference implementation is available[1]. The distribution contains HDL combinatorial logic for the SAES32 instruction (including the S-Boxes) and provisional assembler listings for full AES-128/192/256 and SM4-128.

The package also has C-language emulator code for the instruction logic, "runnable pseudocode" implementations of algorithms, and a set of standards-derived unit tests. This research distribution is primarily intended for obtaining data such as instruction counts and intermediate values but can be readily integrated into many RISC-V cores and emulators.

A. About the AES, SM4 S-Boxes

AES and SM4 can share data paths so it makes sense to explore their additional structural similarities and differences. Both SM4 and AES S-Boxes are constructed from finite field inversion x^{-1} in $\mathsf{GF}(2^8)$ together with a linear (affine) transformations on input and/or output. The inversion makes them "Nyberg S-Boxes" [10] with desirable properties against differential and linear cryptanalysis, while the linear mixing steps are intended to break the bytewise algebraic structure.

Since x^{-1} is an involution (self-inverse) and affine isomorphic regardless of polynomial basis, AES, AES^{-1}, and SM4 S-Boxes really differ only in their inner and outer linear layers.

Boyar and Peralta [4] show how to build low-depth circuits for AES that are composed of a linear top and bottom layers and a shared nonlinear middle stage. Here XOR and XNOR gates are "linear" and the shared nonlinear layer consists of XOR and AND gates only. For this project we created additional top and bottom layers for SM4 that use the same the middle layer as AES and AES^{-1}.

Each S-Box expands an 8-bit input to 21 bits in a linear inner ("top") layer, uses the shared nonlinear 21-to-18 bit mapping as a middle layer, and again compresses 18 bits to 8 bits in the outer ("bottom") layer. Table II gives the individual gate counts to each layer; summing up top, middle, and bottom gives the total S-Box gate count ($\approx$ 128).

Despite such a strict structure and limited choice of gates (that is suboptimal for silicon but very natural to mathematics), these are some of the smallest circuits for AES known. Note that it is possible to implement AES with fewer gates (113 total), but this results in 50% higher circuit depth [3].

[1]AES/SM4 ISA Extension: https://github.com/mjosaarinen/lwaes_isa

TABLE III
RV32 SOC AREA WITH AND WITHOUT SAES32 (AES, AES^{-1}, SM4); "PLUTO" CORE ON AN ARTIX-7 FPGA. EXTAES IS A CPU-EXTERNAL MEMORY-MAPPED AES-ONLY MODULE, PRESENTED FOR COMPARISON.

Resource	**Base**	**SAES32** (Δ)	**EXTAES** (Δ)
Logic LUTs	7,767	8,202 (+435)	9,795 (+2,028
Slice regs	3,319	3,342 (+23)	4,361 (+1,042)
SLICEL	1,571	1,864 (+293)	2,068 (+497)
SLICEM	734	737 (+3)	851 (+117)

TABLE IV
YOSYS SIMPLE CMOS FLOW AREA ESTIMATES FOR SAES32.

Target	GE (NAND2)	Transistors	LTP
AES Encrypt only	642	2,568	25
SM4 Full	767	3,066	25
AES Full	1,240	4,960	28
AES + SM4 Full	1,679	6,714	28

B. Experimental Instruction Encoding and Synthesis

For prototyping we interfaced the SAES32 logic using the *custom-0* opcode and R-type instruction encoding with `fn[4:0]` occupying lower 5 bits of the funct7 field:

[31:30]	[29:25]	[24:20]	[19:15]	[14:12]	[11:7]	[6:0]
00	`fn`	`rs2`	`rs1`	000	`rd`	0001011

The implementation has been tested with PQShield's "Pluto" RISC-V core. We synthesized the same core on low-end Xilinx Artix-7 FPGA target (XC7A35TICSG324-1L) with and without the SAES32 (AES, SM4) instruction extension and related execution pipeline interface. Table III summarizes the relative area added by SAES32. For comparison, we also measured the size of a memory-mapped AES module "EXTAES". This module implements AES encryption only.

Based on our FPGA experiments we estimate that the full (AES, AES^{-1}, SM4) instruction proposal increases the amount of core logic (LUTs) by about 5% over a typical baseline RV32I core, but relatively much less for more complex cores.

Table IV contains area estimates for the SAES32 module (not the additional decoding logic) using the Yosys "Simple CMOS" flow which uses a mock-up ASIC cell library. Here GE is the gate count (NAND2 Equivalents) and Longest Topological Path (LTP) is the reported depth (delay) measure.

Implementors can experiment if it is beneficial to multiplex the S-Box linear layers with the shared middle layer. The required mux logic seems large and increases the circuit depth, so our current reference implementation does not use it.

V. PERFORMANCE AND SECURITY ANALYSIS

The hand-optimized AES implementation[2] referenced in [14] requires 80 core arithmetic instructions per round. The same task can be accomplished with 16 SAES32 instructions. Furthermore, 16 of those 80 are memory loads, which typically require more cycles than a simple arithmetic instruction (or SAES32). Each AES round additionally requires a few operations for loading subkeys and managing instruction flow.

[2]Ko Stoffelen: "RISC-V Crypto" [14] https://github.com/Ko-/riscvcrypto

Overall, based on RV32 and RV64 instruction counts we estimate that the performance of an SAES32 AES can be expected to be more than 500% better than the fastest AES implementations that use the baseline ISA only. Much of the precise performance gain over a table-based implementation depends on the latency of memory load operations.

SAES32-based AES and SM4 implementations are inherently constant-time and resistant to timing attacks. Stoffelen [14] also presents a constant-time, bitsliced AES implementation for RISC-V which requires 2.5 times more cycles than the optimized table-based implementation. So SAES32 speedup over a timing side-channel hardened base ISA implementation is expected to be roughly 15-fold.

We are not aware of any definitive assembler benchmarks for SM4 on RISC-V, but based on instruction count estimates the performance improvement can be expected to be roughly similar or more (over 500 %). Without SAES32 simple SM4 software implementations would benefit from rotation instructions which have been proposed in the RISC-V bit manipulation extension, but are not widely available.

We have only discussed timing side-channel attacks. Since these instructions interact with the main register file, any electromagnetic emission countermeasures would probably have to be extended to additional parts of the CPU core.

VI. Conclusions

We propose a minimalistic RISC-V ISA extension for AES and SM4 block ciphers. The resulting speedup is 500% or more for both ciphers when compared to hand-optimized base ISA assembler implementations that use lookup tables.

In addition to saving energy and reducing latency in secure communications and storage encryption, the main security benefit of the instructions is their constant-time operation and resulting resistance against cache timing attacks. Such countermeasures are expensive in pure software implementations.

The instructions require logic only for a single S-Box, which is combined with additional linear layers for increased code density and performance. The hardware footprint of the instruction is very small as a result. If both AES and SM4 are implemented on the same target they can share data paths which further simplifies hardware. However, AES and SM4 are independent of each other and AES^{-1} is also optional. It is not rare to implement and use the forward AES without inverse AES as common CTR-based AES modes (such as GCM) do not require the inverse cipher for decryption [5].

This proposal is targeted towards (ultra) lightweight MCUs and SoCs. A different type of ISA extension may provide additional speedups on 64-bit and vectorized platforms, but with the cost of increased implementation area. Designers may still want to choose this minimal-footprint option if timing side-channel resistance is their primary concern.

Postscript. Since the original submission and preprint distribution of this work in February 2020, these "Scalar 32-bit" instructions have been adopted to the RISC-V Crypto Extension Proposal [8]. We've changed the instruction names to reflect that specification; thanks to Ben Marshall et al.

References

[1] ARM. Arm A64 Instruction Set Architecture. Armv8, for Armv8-A architecture profile, 2019. ARM DDI 0596 (ID 120619). URL: https://static.docs.arm.com/ddi0595/f/SysReg_xml_v86A-2019-12.pdf.

[2] D. J. Bernstein. Cache-timing attacks on AES. Web-published Manuscript, April 2005. URL: http://cr.yp.to/papers.html#cachetiming.

[3] J. Boyar and R. Peralta. A new combinational logic minimization technique with applications to cryptology. In P. Festa, editor, *Experimental Algorithms, 9th International Symposium, SEA 2010, Ischia Island, Naples, Italy, May 20-22, 2010. Proceedings*, volume 6049 of *Lecture Notes in Computer Science*, pages 178–189. Springer, 2010. doi:10.1007/978-3-642-13193-6_16.

[4] J. Boyar and R. Peralta. A small depth-16 circuit for the AES S-box. In D. Gritzalis, S. Furnell, and M. Theoharidou, editors, *Information Security and Privacy Research - 27th IFIP TC 11 Information Security and Privacy Conference, SEC 2012, Heraklion, Crete, Greece, June 4-6, 2012. Proceedings*, volume 376 of *IFIP Advances in Information and Communication Technology*, pages 287–298. Springer, 2012. URL: https://eprint.iacr.org/2011/332, doi:10.1007/978-3-642-30436-1_24.

[5] M. Dworkin. Recommendation for block cipher modes of operation: Methods and techniques. NIST Special Publication SP 800-38A, December 2001. doi:10.6028/NIST.SP.800-38A.

[6] S. Gueron. Intel's new AES instructions for enhanced performance and security. In O. Dunkelman, editor, *Fast Software Encryption, 16th International Workshop, FSE 2009, Leuven, Belgium, February 22-25, 2009, Revised Selected Papers*, volume 5665 of *Lecture Notes in Computer Science*, pages 51–66. Springer, 2009. doi:10.1007/978-3-642-03317-9_4.

[7] ISO/IEC. Information technology – security techniques – encryption algorithms – part 3: Block ciphers. Amendment 2: SM4. ISO/IEC Standard 18033-3:2010/DAmd 2 (en), 2018.

[8] B. Marshall, editor. *RISC-V Cryptographic Extension Proposals*. August 2020. URL: https://github.com/scarv/riscv-crypto/.

[9] NIST. Advanced Encryption Standard (AES). Federal Information Processing Standards Publication FIPS 197, November 2001. doi:10.6028/NIST.FIPS.197.

[10] K. Nyberg. Differentially uniform mappings for cryptography. In T. Helleseth, editor, *Advances in Cryptology - EUROCRYPT '93, Workshop on the Theory and Application of of Cryptographic Techniques, Lofthus, Norway, May 23-27, 1993, Proceedings*, volume 765 of *Lecture Notes in Computer Science*, pages 55–64. Springer, 1993. doi:10.1007/3-540-48285-7_6.

[11] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of AES. In D. Pointcheval, editor, *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006. URL: https://eprint.iacr.org/2005/271, doi:10.1007/11605805_1.

[12] E. Rescorla. The Transport Layer Security (TLS) protocol version 1.3. IETF RFC 8446, August 2018. doi:10.17487/RFC8446.

[13] SAC. GB/T 32907-2016: SM4 block cipher algorithm. Standardization Administration of China, August 2016. Also GM/T 0002-2012. URL: http://www.gmbz.org.cn/upload/2018-04-04/1522788048733065051.pdf.

[14] K. Stoffelen. Efficient cryptography on the RISC-V architecture. In P. Schwabe and N. Thériault, editors, *Progress in Cryptology - LATINCRYPT 2019 - 6th International Conference on Cryptology and Information Security in Latin America, Santiago de Chile, Chile, October 2-4, 2019, Proceedings*, volume 11774 of *Lecture Notes in Computer Science*, pages 323–340. Springer, 2019. URL: https://eprint.iacr.org/2019/794, doi:10.1007/978-3-030-30530-7_16.

[15] A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*. RISC-V Foundation, December 2019. URL: https://riscv.org/specifications/.