SNEIK on Microcontrollers: AVR, ARMv7-M, and RISC-V with Custom Instructions

Markku-Juhani O. Saarinen

PQShield Ltd.
Prama House, 267 Banbury Road
Oxford OX2 7HT, United Kingdom
mjos@pqshield.com

Abstract. SNEIK is a family of lightweight cryptographic algorithms derived from a single 512-bit permutation. The SNEIKEN authenticated encryption and SNEIKHA hash algorithms are candidates in the ongoing NIST Lightweight Cryptography standardization effort. The SNEIGEN "entropy distribution function" was designed to speed up certain functions in post-quantum and lattice-based public key algorithms. We implement and evaluate SNEIK algorithms on popular 8-bit AVR and 32-bit ARMv7-M (Cortex M3/M4) microcontrollers, and also describe an implementation for the open-source RISC-V (RV32I) Instruction Set Architecture (ISA). Our results demonstrate that SNEIK algorithms usually outperform AES and SHA-2/3 on these lightweight targets while having a naturally constant-time design and significantly smaller implementation footprint. The RISC-V architecture is becoming increasingly popular for custom embedded designs that integrate a CPU core with application-specific hardware components. We show that inclusion of two simple custom instructions into the RV32I ISA yields a radical (more than five-fold) speedup of the SNEIK permutation and derived algorithms on that target, allowing us to reach 12.4 cycles/byte SNEIKEN-128 authenticated encryption performance on PQShield's "Crimson Puppy" RV32I-based SoC. Our performance measurements are for realistic message sizes and have been made using real hardware. We also offer implementation size metrics in terms of RAM, firmware size, and additional FPGA logic for the custom instruction set extensions.

Keywords: SNEIK · SNEIKEN · SNEIKHA · Lightweight Cryptography · AVR · Cortex-M4 · ARMv7-M · RISC-V · ISA Extensions · Crimson Puppy

1 Introduction

There are two traditional design targets for lightweight cryptographic algorithms; compact and efficient stand-alone hardware implementation and software implementation for lightweight CPUs. NSA's SIMON and SPECK algorithms are an example of this duality; SIMON is optimized for hardware while SPECK is optimized for software [BSS+13]. The ongoing NIST Lightweight Cryptography (LWC) standardization effort[1] has candidates that are clearly in either one of these categories, and some that try to meet both targets.

The emergence of open source RISC-V Instruction Set Architecture (ISA) introduces a viable third option where we can consider custom instruction set extensions that maximize total performance return against power consumption and other costs. This also creates an opportunity to address potential security issues such as side-channel leakage in the processor itself. Naturally one would prefer any new instructions to have broad performance benefits for all kinds of cryptographic tasks, rather just for some very specific use cases.

[1] NIST LWC Project: https://csrc.nist.gov/projects/lightweight-cryptography

Version 20190816170000

The NIST evaluation criteria also emphasizes algorithms that can share features of the implementation to do multiple things – namely cryptographic hashing and authenticated encryption – since both of these functions are often required by the same applications.

SNEIK [Saa19c] is one such proposal; its use cases are more complex than simple RFID authentication or verification of a hashed password. The design intends to satisfy *all* symmetric cryptographic needs of cryptographic protocols and applications, including tasks such as pseudorandom number generation and key derivation.

Paper outline and contributions. Section 2 describes the SNEIK family of algorithms, notably the SNEIKEN AEAD and SNEIKHA hash. Section 3 discusses their implementation on an 8-bit AVR microcontroller where our hand-crafted assembly language implementation is roughly 10 times faster than compiled C reference code. SNEIK was largely designed with the 32-bit low-power Cortex M3/M4 target in mind (Section 4), so our new assembler implementation achieves only about 10% performance improvement over the original on that target. Section 5 discusses the RISC-V (RV32I) ISA, which lacks some features that are important to SNEIK, most importantly rotation instructions. We observe that RISC-V embedded targets are ideal for custom instruction set extensions, and propose two simple instructions that significantly speed up (and lower the power consumption) of the SNEIK permutation. These benefits are directly translated to more efficient SNEIKHA hashing, SNEIKEN encryption, and other applications. We compare SNEIK to current NIST algorithms on our target platforms and conclude in Section 6.

2 The SNEIK Family of Algorithms

SNEIK [Saa19c] is a family of lightweight cryptographic primitives whose security is derived from the core SNEIK permutation (Section 2.2). Several SNEIK algorithms are candidates in the NIST Lightweight Cryptography (LWC) standardization effort (See Table 1.)

- SNEIKEN provides Authenticated Encryption with Associated Data (AEAD) functionality and has a similar external interface to the AES-GCM standard [NIS01, Dwo07]. Modern protocols such as TLS 1.3 use AEADs to secure bulk data [Res18].
- SNEIKHA is a cryptographic hash function intended as a replacement for SHA-2/3 [NIS15b, NIS15a] in lightweight applications. It can be used in digital signatures, to hash passwords, etc. It is also an eXtensible Output Function (XOF) like SHAKE.

An appendix of the SNEIK specification further defines SNEIGEN, which is characterized as an "entropy distribution function" – essentially a fast XOF or pseudo-random generator. SNEIGEN is equivalent to SNEIKHA but has only $\rho \in \{3, 4, 5\}$ rounds, corresponding to rate $r \in \{384, 320, 256\}$. SNEIGEN is significantly faster than SNEIKHA but is intended only for use cases where certain types of attacks are not a concern. The "R5Sneik" variant of the Round5 post-quantum public key encryption algorithm [BBF+19, GMZB+19] uses both SNEIKHA and SNEIGEN as its internal building blocks, resulting in significantly increased overall performance on Cortex M4 [Saa19a].

2.1 Permutation-based cryptography

As is usual in permutation-based cryptography, the SNEIKEN state ($b = 512$ bits) is split into r-bit "rate" and c-bit "capacity" parts. The performance of permutation (sponge) modes is mostly determined by number of rounds ρ and the rate r; this is the number of input/output bits processed per permutation invocation.

The capacity parameter $c = b - r$ is related to the security level of the mode – this is the "secret" portion of the state that does not directly interact with input or output.

Table 1: SNEIK Parameters proposed for the NIST LWC Project.

Name	Type	Security	Rate	Rounds
SNEIKEN128	AEAD	2^{128}	$r = 384$	$\rho = 6$
SNEIKEN192	AEAD	2^{192}	$r = 320$	$\rho = 7$
SNEIKEN256	AEAD	2^{256}	$r = 256$	$\rho = 8$
SNEIKHA256	Hash	2^{128}	$r = 256$	$\rho = 8$
SNEIKHA384	Hash	2^{192}	$r = 128$	$\rho = 8$
(SNEIGEN)	(A fast entropy distribution function.)			

For SNEIKEN the confidentiality level is designed to be 2^c (offline computation), while integrity is set at 2^{64} (messages required for successful forgery). For SNEIKHA we claim $2^{c/2}$ security against collision attacks – the security against pre-image attacks may be significantly higher in many use cases.

We do not go into specifics of the BLNK2 sponge modes in this work, but note that they are quite simple and consist almost exclusively of XOR operations of input with words of the internal state ("absorption"), outputting a part of the internal state ("squeezing"), and various padding details. The hash mode of SNEIKHA is virtually equivalent to the Sponge modes used by SHA-3 / SHAKE algorithms, apart from padding details. Little or no storage beyond the 64-byte state is required by the modes.

As the sponge mode wrappers are very simple, the overall performance largely depends on the implementation details of the underlying SNEIK permutation. We note that in addition to the modes that have been proposed for standardization, the BLNK2 framework used by SNEIK supports many additional applications, including full protocol designs [Saa14].

2.2 The SNEIK Permutation $f512$

The SNEIK permutation operates on a $b = 512$ bit state organized as sixteen ($n = 16$) 32-bit words ($\mathsf{s}[0], \mathsf{s}[1], \ldots \mathsf{s}[15]$). Each round has of 16 steps, corresponding to state words.

The permutation is a pure ARX design; it is composed of 32-bit addition ("A" = $\boxplus$), cyclic left rotation ("R" = $\lll$), and exclusive-or operations ("X" = $\oplus$). For purposes of analysis one may view the SNEIK permutation non-linear feedback shift register, and write out the specific substeps t_i:

$$
\begin{aligned}
t_1 &= \mathsf{s}[i-n] \oplus \mathsf{d}[i] \\
t_2 &= t_1 \boxplus \mathsf{s}[i-1] \\
t_3 &= t_2 \oplus (t_2 \lll 24) \oplus (t_2 \lll 25) \\
t_4 &= t_3 \oplus (\mathsf{s}[i-2] \lll 1) \\
t_5 &= t_4 \boxplus \mathsf{s}[i-n+2] \\
t_6 &= t_5 \oplus (t_5 \lll 9) \oplus (t_5 \lll 17) \\
t_7 &= t_6 \oplus \mathsf{s}[i-n+1] \\
\mathsf{s}[i] &= t_7.
\end{aligned}
\tag{1}
$$

The $\mathsf{d}[i]$ input in substep t_1 provides an 8-bit round constant $\mathsf{rc}[i/n]$ when $i \equiv 0 \pmod n$ and a 8-bit domain separator δ when $i \equiv 1 \pmod n$. It is zero for 14 of the $n = 16$ steps. Since both ρ and δ inputs are variable (even in the same higher-level algorithm), SNEIK is actually a "permutation family".

The "feedback" is usually implemented with a cyclic word buffer in software; $\mathsf{s}[i-n]$ is at the same memory location as $\mathsf{s}[i]$. With this type of indexing, we see that computation

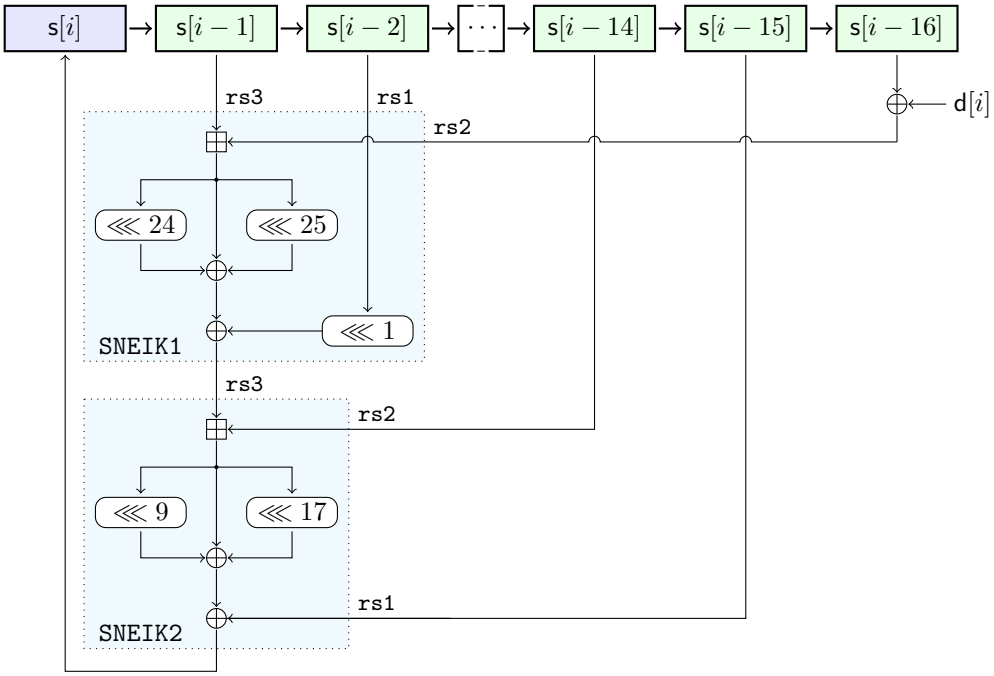

Figure 1: Illustration of a single SNEIK step. Five input words are used to refresh a single output word. The state consists of sixteen 32-bit words, which conveniently fits into the register file of RISC-V. We see that almost the entire permutation is captured by the two custom instructions `SNEIK1` and `SNEIK2`. The constant input $\mathsf{d}[i]$ is zero for most steps.

of each new word uses a cyclically continuous block of five input words: $\mathsf{s}[i] = f_{\mathsf{d}[i]}(\mathsf{s}[i-2], \mathsf{s}[i-1], \mathsf{s}[i], \mathsf{s}[i+1], \mathsf{s}[i+2])$. This sliding window representation ("two words to the left and two words to the right") is essential in most of our implementations since it allows us to keep four state words in registers and load just one ($\mathsf{s}[i+2]$) when computing and storing a new value for $\mathsf{s}[i]$. Word $\mathsf{s}[i-2]$ are not be needed before the next round.

Note. A cryptanalytic flaw in the permutation of SNEIK v1.0 [Saa19b] was identified shortly after its publication [Per19]. The problem is addressed in the current SNEIK v1.1 [Saa19c] proposal. This new version has an additional of a single-bit rotation in sub-step t_4 of Equation 1. Furthermore, the size of the AEAD authentication tag was reduced from 128 bits to 64 bits to save bandwidth.

3 Atmel/Microchip AVR

AVR is a family of 8-bit microcontrollers which are widely used in commercial embedded applications and hobbyist projects. The highly popular and accessible Arduino[2] board and development system is based on AVR. The AVR instruction set [Atm16] is mostly consistent across a large section of Atmel/Microchip microcontrollers.

Most instructions are two bytes long and execute in a single cycle. Instructions are limited to two operands; the destination register doubles as the first source register, which of often necessitates moving data around. The AVR register file consists of 32 byte-wide registers `R0...R31`, with `R1` permanently set to zero. Three fixed pairs of registers can be paired to provide 16-bit indirect access via `X=R27:R26`, `Y=R29:R28`, and `Z=R31:R30`. There is an additional stack pointer `SP` and various single-bit conditional flags such as carry.

[2]The Arduino platform: https://www.arduino.cc/

Table 2: SNEIK size and performance on the 8-bit AVR platform.

256-bit Secret State: **Implementation**	Code **Bytes**	$\rho = 8$ **Cycles**	$r = 256$ **Cycles/B**
AVR Fast (Asm)	2,144	9,265	289.5
AVR Small (Asm)	666	9,648	301.5
AVR Ref C (-Os)	4,326	99,281	3,102.5
AVR Ref C (-Ofast)	5,178	101,574	3,174.2

Message size (bytes) and corresponding cycles:

Algorithm	**50B**	**100B**	**500B**	**1,000B**
SNEIKEN128	29,761	37,659	100,834	179,805
SNEIKEN192	34,365	43,415	132,016	246,816
SNEIKEN256	39,028	58,483	177,107	334,640
SNEIKHA256	19,852	38,943	154,682	308,603
SNEIKHA384	57,025	85,409	321,734	614,829

Implementation. We compiled the SNEIK reference C implementation using GCC (`avr-gcc` 5.4.0 on Ubuntu 18.04) to the AVR target and instrumented it with cycle ("tick") counters. After verifying correct operation we left the main mode wrappers (`encrypt.c` and `hash.c`) largely intact, only changing the length arguments to recommended `size_t` type from `unsigned long long`.

We note that the hash wrapper `hash.o` is 288 bytes, and `encrypt.o` provides AEAD encryption, decryption, and authentication functionality in 1032 bytes. These are the only components required (in addition to the shared permutation) to implement and use SNEIKHA and SNEIKEN in applications.

Assembler optimization of permutation. To translate the ARX arithmetic to AVR instruction set we note that a 32-bit XOR can be implemented with four `EOR` instructions. A 32-bit addition is one `ADD` and three `ADC` (add with carry) instructions. AVR has instructions only for 1-bit shifts; the `ROR` and `ROL` rotation instructions actually insert the carry bit as the most or least significant bit of the byte rather than performing a plain 8-bit cyclic rotation.

To implement a 32-bit left rotation ($x \lll 1$) we use a single `LSL` logical shift left (which sets the carry), followed by three `ROL` carry-utilizing shifts on bytes of increasing significance, and finally, an `ADC` that puts the wraparound bit back to the first byte with the aid of `R1`, which always holds zero by convention.

We adopt the sliding window approach and group twenty 8-bit registers into five 32-bit "meta-registers" `W0`, `W1`, `W2`, `W3`, `WT`. This still leaves `R20` for loop counter, `R21` for δ (one byte is sufficient), a pointer `X` for the round constants, and `Z` for the state. Each meta-register assignment uses a continuous set of registers, which helped to simplify macros:

```
W0 = R07:R06:R05:R04    W1 = R11:R10:R09:R08    W2 = R15:R14:R13:R12
W3 = R19:R18:R17:R16    WT = R25:R24:R23:R22
```

Rotation by a multiple of 8 bits is just a matter of shuffling or "renaming" the individual registers used to represent the number. We observe from Equation 1 that all of the required left rotations $(24, 25, 1, 9, 17)$ are either by $8n$ or $8n + 1$ bits and therefore realizable by no more than five instructions; one can also directly exclusive-or the contents of a meta-register rotated by 24 bits over with four `EOR` operations. The SNEIK rotations were specifically chosen to have this property; we knew that rotating a word by 4 bits is four times more expensive than by 9, 17, or 25 bits on AVR!

We created GNU Assembler macros that implement a single SNEIK step in about 63 AVR instructions, including one 4-byte load and one 4-byte store. These were then used to construct two versions of the overall permutation:

- **"Fast"** (`sneik_f512_avr_fast.S`): unrolled round, $1145 \times \rho + 105$ cycles per call.
- **"Small"** (`sneik_f512_avr_small.S`): unrolls 4 steps, $1191 \times \rho + 120$ cycles per call.

Performance. Table 2 summarizes our AVR results. We recommend using the "small" version in most applications as its performance penalty is not very large. The permutation and modes use just a few bytes of (stack) RAM in addition to the 64-byte state.

Benchmarks were measured on a 16 MHz ATmega2560 chip (Arduino MEGA 2560 compatible board). The numbers for `-Ofast` and `-Os` optimization flags are not flipped; this is a gcc quirk. We offer timings for the raw permutation as well as SNEIKEN and SNEIKHA when operating on messages of various sizes, following the style of [BEE+12]. In this paper the performance numbers given for SNEIKEN are an average of successful encapsulation (authenticated encryption) and decapsulation (authenticated decryption) operations with zero-length associated data.

4 ARMv7-M (Cortex M3/M4)

The ARMv7-M microcontroller architecture [ARM18] is one of three ARMv7 architecture profiles, the other two being the ARMv7-A application profile (e.g. 32-bit Android phones) and the ARMv7-R real-time profile. There is a common subset of Thumb (16-bit) instructions among all the three; ARMv7-A/R profiles also support 32-bit instructions.

ARMv7-M is the most lightweight of the three in terms of on-chip area, power consumption, and also price. It is implemented in the Cortex-M3 and M4 cores used by a large number of microcontroller vendors: NXP, ST, TI, Silicon Labs, Nordic, and others.

SecurCore SC300 - based SIMs, smart cards, and other security elements are also based on Cortex M3 and use the ARMv7-M ISA. STMicroelectronics alone has reported to have shipped more than one billion such ST33 (SC300) units by early 2019 [STM19].

Note that ARMv7-M is a Harvard (dual-bus) architecture and differs significantly from ARMv6-M (Cortex-M0 and Cortex-M1), a Von Neumann (single bus) architecture.

C Implementation. We started our project as with AVR; compiling the SNEIK reference implementations on target with a C cross compiler, which in this case was a version of GCC 8.2.1 released by ARM (`arm-none-eabi-gcc`, 8-2018-q4-major).

We again implemented full hash and AEAD functionality, not just the permutation. The outputs are compliant with SNEIKEN and SNEIKHA test vectors. The hash and AEAD mode implementations (i.e. excluding the permutation) with `-Os` optimization level are very compact: `hash.o` is 180 bytes and `encrypt.o` is 672 bytes. We note that the C optimization level of the mode implementations affects overall performance by about 5-15%; however the speed-optimized variants can be two or three times larger.

Assembler Optimization. SNEIK was designed to be particularly effective on this instruction set; in addition to a 32-bit datapath, ARMv7 allows a second source operand to be rotated "for free" in most arithmetic operations, including XOR. This significantly reduces the number of instructions required for the SNEIK permutation.

Unfortunately ARMv7 has only of 13+2 general purpose registers (LR and even SP can be used with certain caveats), so it is not possible to keep the entire SNEIK state in the register file. There are additional implementation considerations as only the lower eight registers are truly general purpose; `R8-R12` are not accessible by some Thumb instructions.

The core of the implementation is built form two assembler macros. `SNEIK1` and `SNEIK2` perform steps $t_2 \cdots t_4$ and $t_5 \cdots t_7$ of Equation 1, respectively:

```
.macro SNEIK1  x, y, z  // x=s[i], y=s[(i+14)%16], z=s[(i+15)%16]
    ADD  \x, \x, \z             // x += z;
    EOR  R3, \x, \x, ROR #8     // x ^= (x <<< 24)
    EOR  \x, R3, \x, ROR #7     //     ^ (x <<< 25);
    EOR  \x, \x, \y, ROR #31    // x ^= (y <<< 1);
.endm
```

```
.macro SNEIK2  x, y, z  // x=s[i], y=s[(i+1)%16], z=s[(i+2)%16]
    ADD  \x, \x, \z             // x += z;
    EOR  R3, \x, \x, ROR #23    // x ^= (x <<< 9)
    EOR  \x, R3, \x, ROR #15    //     ^ (x <<< 17);
    EOR  \x, \x, \y             // x ^= y;
.endm
```

Both of these macros consist of four instructions; 8 total. One additional load and/or one store operation *may* be required for each step, and each round requires few instructions for domain, round constant, and looping. In experimentation with real hardware we consistently obtained $179 \times \rho + 50$ cycles per function invocation; 11.2 average per step.

In our "ARMv7 Fast" implementation the registers `R0`, `R1`, `R2`, and `R12` are taken up by state pointer `*s`, domain separator δ, round counter, and round constant table `*rc`, respectively. We use `R3` as a temporary register and assign `R4`-`R7` as our sliding window, but keep *some* state words permanently in registers; `R8`-`R11` and `R14` (=`LR`) hold the state words `s[2...6]` between iterations. The remaining 11 words need to stored and loaded from memory in each round.

The difference between a hand-crafted assembler and compiled C implementations is not 10-fold like with AVR. The SNEIK primitive operations are quite natural to the Thumb instruction set. We looked at the code created by the compiler and observed that it nicely utilizes the special rotation instructions – even though those are expressed in terms of shifts in the C implementation – and yields a relatively good performance from the unmodified "opt" reference implementation. Again, the `-Os` compiler flag strangely yields better performance than `-Ofast` for the core permutation (but not for the modes!).

Cortex M3 and M4 support unaligned access, but this may not be safe across all ARMv7-M systems. Some performance gain was obtained by changing the `encrypt.c` code to process full blocks as 32-bit words. This change did strangely had a negative performance impact on `hash.c`. Some compilers may perform this optimization automatically since the block size is always of full word length.

Performance. Table 3 summarizes our ARMv7-m measurements. Again, the cycle count for SNEIKEN is an average of successful encapsulation and decapsulation operations (which have very similar performance profile). Implementations were benchmarked on a STM32F407VGT6 Cortex M4 microcontroller from STMicroelectronics (STM32F4 discovery development board[3]). We instrumented the code with cycle-accurate counters ("SysTick") available on the microcontroller itself. The MCU was clocked at 24 MHz, which helps to remove wait states caused by the flash ROM. A serial interface was used to read out the results. Note that there is some performance variation across ARMv7-m implementations from different vendors and Cortex M3 tends to be 1-2% faster than M4. The RAM usage of the implementation is less than 100 bytes.

[3]STM32F4 kit used: https://www.st.com/en/evaluation-tools/stm32f4discovery.html

Table 3: SNEIK size and performance on Cortex M4 (ARMv7-M).

256-bit Secret State: **Implementation**	Code **Bytes**	$\rho = 8$ **Cycles**	$r = 256$ **Cycles/B**
ARMv7 Fast (Asm)	592	1,482	46.3
ARMv7 Small (Asm)	240	1,814	56.7
ARMv7-M Ref C (-Os)	656	1,715	53.6
ARMv7-M Ref C (-Ofast)	748	1,635	51.1

	Message size (bytes) and corresponding cycles:			
Algorithm	**50B**	**100B**	**500B**	**1,000B**
SNEIKEN128	4,866	6,118	16,146	28,681
SNEIKEN192	5,653	7,156	21,106	39,010
SNEIKEN256	6,449	9,397	28,197	52,897
SNEIKHA256	3,268	6,293	24,785	49,366
SNEIKHA384	9,498	14,288	54,313	103,953

5 RISC-V

The RISC-V Instruction Set Architecture (ISA) is an increasingly popular open-source alternative to commercial vendor-defined ISAs (e.g. ARM, Intel, MIPS). The specifications are published by the RISC-V Foundation[4], and allow anyone to create royalty-free implementations. The RISC-V ISA is now well supported by open source toolchains (notably the GNU C compiler and libraries) and operating systems such as Linux.

Due to its healthy development ecosystem, an increasing number of commercial vendors are using the ISA: Rambus, Nvidia, Western Digital, and SiFive are among companies that have either released or announced products based on RISC-V. PQShield has developed RISC-V cores and systems specifically for cryptography and security applications.

The RISC-V ISA comes in many shapes, the main ones being RV32I and RV64I, the 32- and 64-bit base variants. There are numerous optional extensions which add to the capabilities of the base set. At the time of writing, seven have been ratified: Multiplication and division instructions ("M"), atomic instructions ("A"), control and status registers ("Zicsr"), single-, double, and quad-precision floating point arithmetic ("F", "D", "Q"), and 16-bit compressed instructions ("C"). The specification even allocates specific code points for custom instructions.

However, all variants are intended to be backwards compatible with the base ISA, and processor designers may choose to emulate optional instructions in software. We will focus on the (smallest) RV32I variant and provide implementation-independent instruction counts in addition to cycle counts on our RV32I hardware implementation.

5.1 SNEIK Permutation on "Base" RV32I

The RV32I architecture has an exceptionally large register file with 31 general-purpose 32-bit registers[5]. This allows us to fit the entire 16-word state of SNEIK into the registers, removing the need for load and store operations between rounds. This is a very large save, especially as RISC-V generally does not allow one to combine load and store operations with arithmetic. On a negative side, the base instruction set does *not* provide instructions for bit rotations; those are only provided by the (draft) "B" Bit Manipulation extension.

[4]RISC-V Foundation, ISA specifications: https://riscv.org/
[5]The register file is reduced to fifteen (plus zero) in the RV32E variant.

```
// A SNEIK Step. Window S[0] = f(S[0], S[1], S[2], S[-2], S[-1]).

.macro SNEIKS  a, b, c, d, e
    ADD   \a, \a, \e        // a += e;  ( a=s[i], e=s[(i+15)%16] )
    SLLI  a2, \a, 24        // a ^= (a <<< 24)
    SRLI  s6, \a,  8
    XOR   a2, a2, s6
    SLLI  s6, \a, 25        //    ^ (a <<< 25);
    XOR   a2, a2, s6
    SRLI  s6, \a,  7
    XOR   a2, a2, s6
    XOR   \a, \a, a2
    SLLI  a2, \d,  1        // a ^= (d <<< 1);  ( d=s[(i+14)%16] )
    SRLI  s6, \d, 31
    XOR   a2, a2, s6
    XOR   \a, \a, a2
    ADD   \a, \a, \c        // a += c;          ( c=s[(i+ 2)%16] )
    SLLI  a2, \a,  9        // a ^= (a <<< 9)
    SRLI  s6, \a, 23
    XOR   a2, a2, s6
    SLLI  s6, \a, 17        //    ^ (a <<< 17);
    XOR   a2, a2, s6
    SRLI  s6, \a, 15
    XOR   a2, a2, s6
    XOR   \a, \a, a2
    XOR   \a, \a, \b        // a ^= b;          ( b=s[(i+ 1)%16] )
.endm
```

Listing 1: A SNEIK step on RV32I. The five macro inputs constitute a "sliding window"; $(a, b, c, d, e) = (\mathtt{s}[\, i\,], \mathtt{s}[\, i \boxplus 1\,], \mathtt{s}[\, i \boxplus 2\,], \mathtt{s}[\, i \boxplus 14\,], \mathtt{s}[\, i \boxplus 15\,])$. This sequence of 23 instructions is replaced by just two custom R4-type instructions `SNEIK1` and `SNEIK2` in this work. With RV64I and other 64-bit ISAs we can potentially create a single 3-input ($3 \times 64 \rightarrow 64$ - bit) instruction that computes two full steps, a further four-fold improvement.

Each of the five left rotation operations contained in a single step (Equation 1) will have to be implemented with one n-bit left shift, one $32 - n$ bit right shift, and a combination operation. So a single step has two `ADD`s, 11 `XOR`s, and five `SLLI` and `SRLI`s shifts each, totaling 23 instructions as shown in Listing 1.

Two registers, `a2` (`X11`) and `s6` (`X24`), are used as temporary variables in the macro. The 16-word state is kept in registers `t0`-`a6`, `a3`-`a7`, and `s0`-`s3`. In addition to the $23 \times 16 = 368$ instructions for stepping, an additional five are needed for fetching and `XOR`ing the round constant, domain δ, incrementing the loop counter, and a conditional branch, bringing the total to 373 instructions per round (of which 3 can be eliminated by unrolling fully). Loading and saving the state, setting up pointers, and handling stack adds a 55 instruction overhead, bringing the total to $373 \times \rho + 55$ instructions.

Performance. We cross-compiled the SNEIK reference code with RISC-V GCC 8.2.0 (`-march=rv32i`), and executed it with PQShield's "`pqse`" emulator to obtain platform-independent instruction counts; see Table 5 for results. The compiler is actually able to do a "perfect" $373 \times \rho$ - instruction job on the permutation (with `-Os`), although the function call overhead is higher, 73 instructions. A size-optimized SNEIKHA hash mode `hash.o` is 336 bytes, while a SNEIKEN `encrypt.o` is 1096 bytes. RAM usage of the implementation is approximately 128 bytes, including the 64-byte state.

5.2 Instruction Set Extension

A single SNEIK step splits conveniently into two parts, which both take in three 32-bit input values and output one (Figure 1). This is ideal for a simple instruction set extension. We name our custom 32-bit instructions `SNEIK1` and `SNEIK2`. They have the same purpose as the assembler macros of the same name in our ARMv7-m implementation (Section 4). We can consider various implementation options:

- On architectures that only allow two source operands, one may use simplified two-input (R-type) versions of `SNEIK1` and `SNEIK2` that need two "external" `ADD` instructions can per step. Each step then has four instructions, rather than just two. If the problem lies in instruction encoding (e.g. 16-bit instructions), we note that the output register is essentially always one of the inputs (`rd` = `rs3`).

- On RV64I and other 64-bit targets *two full steps* can be expressed as a single three-input instruction ("the `SNEIK` instruction") operating on $3 \times 64 = 192$ input bits and producing 64. This reduces the instruction count further by a factor of 4. Only eight registers are needed to hold the state in this case – which also applies to the 64-bit double-precision floating point register file of the "D" extension.

- In an unrolled round, one of the source operands to `SNEIK1` and `SNEIK2` is always the result of the immediately preceding operation. Therefore it is available in the pipeline unless an interrupt occurs; a CPU designer can impose an penalty of several cycles for this very rarely occurring register file fetch.

5.3 Experiments: Crimson Puppy

We use the "Crimson Puppy" RISC-V core developed by PQShield Ltd. in our testing and experiments. This core is notable for its small size and Harvard architecture (like AVR and Cortex-M3/4). Crimson Puppy achieves single-cycle instruction execution with about 95% probability in practice. All ALU operations are single-cycle, while jumps, branches, loads, and store operations may require 1-2 additional cycles.

The core implements the ratified RV32I v2.0 user-level ISA [WA19], which we run in "machine mode" of the privileged ISA. This corresponds to a typical embedded microcontroller set-up. The complexity of this core is somewhere between AVR and Cortex M0 architectures, or even less.

The RISC-V ISA specification does not address implementation details such as instruction timing at all, and cycle counts vary widely from one implementation to another. Since some RISC-V cores require three or more cycles per instruction, we also report the (implementation independent) instruction count for core SNEIK primitives.

Proving the design on FPGA. Hardware synthesis was targeted on the Artix-7 XC7A35T FPGA chip of the Arty7-35T[6] development board. Our basic SoC design used in this project has rudimentary UART input and output peripherals for testing. See Table 4 for a hardware utilization summary for the SoC using Vivado 2019.1. Utilization of all resources except BRAM is well under 10% on this low-end FPGA model.

The SoC makes most of the 36kB BRAM blocks available to the CPU as working memory but SNEIK itself requires a minimal amount RAM. SNEIK implementation was tested on a relatively feature-rich runtime environment that provides standard C libraries (for the NIST testbench) and therefore needs some working RAM.

[6] Digilent Arty A7-35T: https://www.xilinx.com/products/boards-and-kits/arty.html

Table 4: Artix-7 (XC7A35T) resource utilization of the "Crimson Puppy" RV32I SoC with and without SNEIK instructions.

Synthesis Target	LUT	FF	Slice	DSP	MHz
Base RV32I SoC	1,547	460	493	0	100.0
..with 3-input SNEIK	1,805	484	558	0	100.0
Change	+258	+24	+65	+0	−0.0

Extending the ISA. The "Crimson Puppy" core has an option for R4-type instructions with three source register operands `rs1`, `rs2`, and `rs3`. There may be a penalty if none of the three source operands is in the pipeline (as a result of immediately preceding arithmetic) and needs to be fetched from the register file. We also tested the 2-input versions that use more typical R-type encoding and data paths but the difference in terms of implementation size and timing turned out to be negligible.

We use the *custom-0* major opcode of the ISA specification [WA19] to define a straightforward encoding for our new R4-type instructions as follows:

[31:27]	[26:25]	[24:20]	[19:15]	[14:12]	[11:7]	[6:0]	Bitfields
`rs3`	00	`rs2`	`rs1`	001	`rd`	0001011	`SNEIK1`
`rs3`	00	`rs2`	`rs1`	010	`rd`	0001011	`SNEIK2`

For 2-input variants we simply set `rs3` to zero (00000) – this corresponds to R-type encoding with "funct7" set to zero and allows downward compatibility in software – and explains the slightly counter-intuitive input operand ordering that we use.

These new instructions are relatively simple to implement. We may describe their combinatorial logic in "Pseudo Verilog":

```
// rs1_w, rs2_w, rs3_w  have been decoded and fetched

wire [31:0] t_w = rs2_w + rs3_w;          //  t = rs2 + rs3;

wire [31:0] sneik1_w = t_w ^              //  sneik1 = t
            { t_w[7:0], t_w[31:8] } ^     //       (t <<< 24) ^
            { t_w[6:0], t_w[31:7] } ^     //       (t <<< 25) ^
            { rs1_w[30:0], rs1_w[31] };   //       (rs1 <<< 1);

wire [31:0] sneik2_w = t_w ^              //  sneik2 = t ^
            { t_w[22:0], t_w[31:23] } ^   //       (t <<<  9) ^
            { t_w[14:0], t_w[31:15] } ^   //       (t <<< 17) ^
            rs1_w;                        //       rs1;
```

The actual implementation on the "Crimson Puppy" core was not much more complicated than this; about a dozen lines of Verilog HDL. For the two-input versions, one can simply assign `t_w` = `rs2_w` directly.

It is clear that the above additional circuitry would not be very substantial as a separate module. However it makes more sense to evaluate it as a part of the whole core. From Table 4 we see that the two custom instructions add no more than 258 LUTs and 65 Slices to the size of "Crimson Puppy" on Artix-7. The timings of all synthesized variants is within 0.5ns of each other, and the same operating frequency suits them all.

Performance results for the 2-input variant are not included in Table 5, but we note that $\rho = 8$ permutation was 695 or $119 + 72 \times \rho$ cycles. Hashing 1000 B with SNEIKHA required 25,463 cycles, 47 % more than the 3-input version, but it is still a four times faster than with plain RV32I.

Table 5: SNEIK cycle count with and without 3-input SNEIK1 and SNEIK2 custom instructions on "Crimson Puppy" – and instruction counts for plain RV32I.

256-bit Secret State:		Code	$\rho = 8$	$r = 256$
Implementation		**Bytes**	**Cycles**	**Cycles/B**
Custom	Asm	384	439	13.7
Plain	Asm	1,728	3,134	97.9
# Instr.	Asm	*same*	3,039	95.0
# Instr.	Ref C (-Os)	1,736	3,057	95.5
# Instr.	Ref C (-Ofast)	1,772	3,304	103.3

		Input size (bytes) and corresponding instructions or cycles:			
Mode	**Target**	**50B**	**100B**	**500B**	**1,000B**
SNEIKEN128	Custom	2,219	2,755	7,043	12,403
SNEIKEN128	Plain	10,311	12,870	33,342	58,932
SNEIKEN128	# Instr.	9,629	12,006	31,022	54,792
SNEIKEN192	Custom	2,548	3,238	8,433	14,831
SNEIKEN192	Plain	11,984	15,033	43,818	80,883
SNEIKEN192	# Instr.	11,215	14,027	41,247	76,438
SNEIKEN256	Custom	2,841	3,665	10,309	18,601
SNEIKEN256	Plain	13,621	19,835	58,819	110,231
SNEIKEN256	# Instr.	12,785	18,827	55,979	105,215
SNEIKHA256	Custom	1,331	2,382	8,776	17,271
SNEIKHA256	Plain	6,721	13,162	51,896	103,511
SNEIKHA256	# Instr.	6,338	12,480	49,418	98,648
SNEIKHA384	Custom	3,611	5,517	21,217	40,660
SNEIKHA384	Plain	19,781	29,772	112,847	215,835
SNEIKHA384	# Instr.	18,741	28,107	106,057	202,705

6 Discussion and Conclusions

We have implemented SNEIK algorithms on three (Fig. 2) common lightweight embedded platforms: 8-bit AVR, 32-bit ARMv7-M used in Cortex M3 and M4 cores, and RV32I open source ISA. What is noteworthy is that SNEIKEN (AEAD) and SNEIKHA (a hash) share a majority of their code, resulting in code (ROM) and state size (RAM) savings.

On AVR we showed how to implement the permutation in assembler in a way that is roughly 10 times faster than the compiled C version. We obtain sustained 180 cycles/byte for authenticated encryption of 1kB messages with SNEIKEN128, while plain AES-128 (with precomputed subkeys) requires 213 cycles/byte and more memory [DCK+18]. SNEIKHA256 can hash 500 B in 155k cycles, while SHA2-256 requires 266k cycles, and SHA3-256 716k cycles [BEE+12].

Our new ARMv7-M (Cortex M4) implementation archives 10% performance improvement by using additional registers to store state words across rounds. We demonstrate authenticated encryption of 1kB messages at 28.6 cycles/byte with SNEIKEN128, while the fastest "unprotected" AES-128-CTR implementation in [SS16] requires 34.7 cycles/byte. SNEIKHA256 hashes 1kB messages at 49.3 cycles/byte on ARMv7-M, while the fastest known SHA3 implementation (XKCP[7] distribution) requires 95.3 cycles/byte [Saa19a].

On RISC-V (RV32I) SNEIKEN128 achieves authenticated encryption of 1kB messages at 54.8 instructions/byte, which approximately matches the 57.0 cycles/byte for plain

[7] XKCP, eXtended Keccak Code Package: https://github.com/XKCP/XKCP

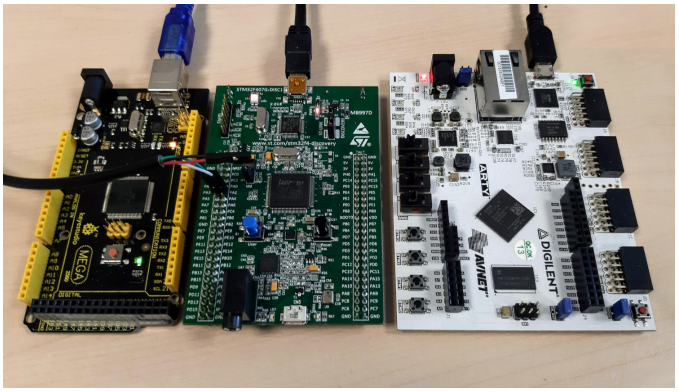

Figure 2: A "family portrait" of the development boards used in this work. From left: An Arduino compatible ATmega2560 board (AVR Architecture), STM32F407-Discovery (Cortex M4, ARMv7-M Architecture), and Arty7-35T XC7A35T (An Artix-7 FPGA used to run our RISC-V RV32I+ ISA core).

AES-128 on a comparable target [Sto19]. SNEIKHA256 requires 98.6 instr/byte for 1kB messages on RV32I, while peak throughput of SHA3-256 is very similar at 101.3 cycles/byte ($r = 1088$, estimated from the reported 13,774 tick permutation [Sto19]).

The RV32I base instruction set lacks rotation instructions, which are dominant in the SNEIK permutation function, resulting in effective halving of per-cycle throughput in comparison to ARMv7. However as an open source architecture, RISC-V may more readily be implemented as a customizable IP core, which motivated us to consider custom ISA extensions. We have discovered that the structure of the SNEIK permutation is highly suitable for this type of optimization, both on lightweight (32-bit RV32I) and 64-bit ISAs.

Our custom RV32I extensions have only a 258 LUT / 65 slice impact on FPGA resource utilization, but speed up the SNEIK permutation by a factor of 7. Our "Crimson Puppy" RISC-V SoC achieves 12.4 cycle/byte SNEIKEN128 performance and 17.3 cycles/byte for SNEIKHA256 when tested on Artix-7 FPGA hardware.

We conclude that SNEIK is well suited for these microcontroller targets, usually outperforming NIST algorithms, while also having a smaller implementation footprint and a naturally constant-time design. A very simple RISC-V instruction set extension easily gives it five-fold speedup (and a similar reduction in energy consumption), which is a great benefit in applications such as custom security controllers.

References

[ARM18] ARM. *ARMv7-M Architecture Reference Manual.* ARM Ltd., 2018. ARM DDI 0403E.d. URL: https://static.docs.arm.com/ddi0403/e/DDI0403E_d_armv7m_arm.pdf.

[Atm16] Atmel. *AVR Instruction Set Manual.* Atmel/Microchip, 2016. Atmel 0856L. URL: http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf.

[BBF+19] Hayo Baan, Sauvik Bhattacharya, Scott R. Fluhrer, Óscar García-Morchón, Thijs Laarhoven, Ronald Rietman, Markku-Juhani O. Saarinen, Ludo Tolhuizen, and Zhenfei Zhang. Round5: Compact and fast post-quantum public-key encryption. In Jintai Ding and Rainer Steinwandt, editors,

Post-Quantum Cryptography - 10th International Conference, PQCrypto 2019, Chongqing, China, May 8-10, 2019 Revised Selected Papers, volume 11505 of *Lecture Notes in Computer Science*, pages 83–102. Springer, 2019. doi:10.1007/978-3-030-25510-7_5.

[BEE+12] Josep Balasch, Baris Ege, Thomas Eisenbarth, Benoît Gérard, Zheng Gong, Tim Güneysu, Stefan Heyse, Stéphanie Kerckhof, François Koeune, Thomas Plos, Thomas Pöppelmann, Francesco Regazzoni, François-Xavier Standaert, Gilles Van Assche, Ronny Van Keer, Loïc van Oldeneel tot Oldenzeel, and Ingo von Maurich. Compact implementation and performance evaluation of hash functions in attiny devices. In Stefan Mangard, editor, *Smart Card Research and Advanced Applications - 11th International Conference, CARDIS 2012, Graz, Austria, November 28-30, 2012, Revised Selected Papers*, volume 7771 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 2012. doi:10.1007/978-3-642-37288-9_11.

[BSS+13] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK families of lightweight block ciphers. IACR Cryptology ePrint Archive 2013/404, June 2013. URL: https://eprint.iacr.org/2013/404.

[DCK+18] Daniel Dinu, Yann Le Corre, Dmitry Khovratovich, Léo Perrin, Johann Großschädl, and Alex Biryukov. Triathlon of lightweight block ciphers for the internet of things. *Journal of Cryptographic Engineering*, page To appear., 2018. URL: https://eprint.iacr.org/2015/209, doi:10.1007/s13389-018-0193-x.

[Dwo07] Morris Dworkin. Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC. NIST Special Publication SP 800-38D, November 2007. doi:10.6028/NIST.SP.800-38D.

[GMZB+19] Oscar Garcia-Morchon, Zhenfei Zhang, Sauvik Bhattacharya, Ronald Rietman, Ludo Tolhuizen, Jose-Luis Torre-Arce, Hayo Baan, Markku-Juhani O. Saarinen, Scott Fluhrer, Thijs Laarhoven, and Rachel Player. Round5. Round 2 submission to the NIST PQC Project, March 2019. URL: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-2-Submissions.

[NIS01] NIST. Advanced Encryption Standard (AES). Federal Information Processing Standards Publication FIPS 197, November 2001. doi:10.6028/NIST.FIPS.197.

[NIS15a] NIST. Secure hash standard (SHS). Federal Information Processing Standards Publication FIPS 180-4, August 2015. doi:10.6028/NIST.FIPS.180-4.

[NIS15b] NIST. SHA-3 standard: Permutation-based hash and extendable-output functions. Federal Information Processing Standards Publication FIPS 202, August 2015. doi:10.6028/NIST.FIPS.202.

[Per19] Léo Perrin. Probability 1 iterated differential in the SNEIK permutation. IACR Cryptology ePrint Archive 2019/374, April 2019. URL: https://eprint.iacr.org/2019/374.

[Res18] Eric Rescorla. The transport layer security (TLS) protocol version 1.3. IETF RFC 8446, August 2018. doi:10.17487/RFC8446.

[Saa14] Markku-Juhani O. Saarinen. Beyond modes: Building a secure record protocol from a cryptographic sponge permutation. In Josh Benaloh, editor, *Topics in Cryptology - CT-RSA 2014 - The Cryptographer's Track at the RSA Conference 2014, San Francisco, CA, USA, February 25-28, 2014. Proceedings*, volume 8366 of *Lecture Notes in Computer Science*, pages 270–285. Springer, 2014. doi:10.1007/978-3-319-04852-9_14.

[Saa19a] Markku-Juhani O. Saarinen. Exploring NIST LWC/PQC synergy with R5Sneik: How SNEIK 1.1 algorithms were designed to support Round5. Second PQC Standardization Conference, 24-25 August 2019, UCSB Campus, USA., August 2019. URL: https://eprint.iacr.org/2018/1116.

[Saa19b] Markku-Juhani O. Saarinen. SNEIK. Round 1 submission to the NIST LWC Project, March 2019. URL: https://csrc.nist.gov/Projects/Lightweight-Cryptography/Round-1-Candidates.

[Saa19c] Markku-Juhani O. Saarinen. SNEIKEN and SNEIKHA v1.1: Authenticated encryption and cryptographic hashing. Technical report, PQShield Ltd., May 2019. URL: https://github.com/pqshield/sneik.

[SS16] Peter Schwabe and Ko Stoffelen. All the AES you need on Cortex-M3 and M4. In Roberto Avanzi and Howard M. Heys, editors, *Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers*, volume 10532 of *Lecture Notes in Computer Science*, pages 180–194. Springer, 2016. doi:10.1007/978-3-319-69453-5_10.

[STM19] STMicroelectronics. STMicroelectronics breaks major milestone for ST33 secure chips with over one billion sold to secure the connected world. Press Release, February 2019. URL: https://www.st.com/content/st_com/en/about/media-center/press-item.html/t4136.html.

[Sto19] Ko Stoffelen. Efficient cryptography on the RISC-V architecture. To appear in Latincrypt 2019 proceedings. IACR Cryptology ePrint Archive 2019/794, July 2019. URL: https://eprint.iacr.org/2019/794.

[WA19] Andrew Waterman and Krste Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20190608-Base-Ratified*. RISC-V Foundation, June 2019. URL: https://riscv.org/specifications/.