NaCl on 8-bit AVR Microcontrollers

Michael Hutter[1] and Peter Schwabe[2] *

[1] Graz University of Technology
Institute for Applied Information Processing and Communications (IAIK)
Inffeldgasse 16a, A-8010, Graz, Austria
michael.hutter@iaik.tugraz.at
[2] Radboud University Nijmegen
Digital Security Group
PO Box 9010, 6500GL Nijmegen, The Netherlands
peter@cryptojedi.org

Abstract. This paper presents first results of the Networking and Cryptography library (NaCl) on the 8-bit AVR family of microcontrollers. We show that NaCl, which has so far been optimized mainly for different desktop and server platforms, is feasible on resource-constrained devices while being very fast and memory efficient. Our implementation shows that encryption using Salsa20 requires 277 cycles/byte, authentication using Poly1305 needs 211 cycles/byte, a Curve25519 scalar multiplication needs 22 954 657 cycles, signing of data using Ed25519 needs 23 211 611 cycles, and verification can be done within 32 937 940 cycles. All implemented primitives provide at least 128-bit security, run in constant time, do not use secret-data-dependent branch conditions, and are open to the public domain (no usage restrictions).

Keywords: Elliptic-curve cryptography, Edwards curves, Curve25519, Ed25519, Salsa20, Poly1305, AVR, ATmega.

1 Introduction

This paper describes implementations of the Networking and Cryptography library (NaCl) [4] on 8-bit AVR microcontrollers. More specifically, we describe two different approaches, one aiming at higher speed, one aiming at smaller memory requirements, of porting NaCl to the AVR ATmega family of microcontrollers. The aim of the high-speed implementation is not to achieve the highest possible speed at all (memory-)costs for all primitives. Similarly, the aim of the low-memory implementation is not to obtain the smallest possible footprint without any performance considerations. The two implementations are rather two example tradeoffs between speed and memory footprint that we consider reasonable and useful for various applications and different microcontrollers in the ATmega family.

* Part of this work was done while Peter Schwabe was employed by the Research Center for Information Technology Innovation, Academia Sinica, Taiwan. Permanent ID of this document: cd4aad485407c33ece17e509622eb554. Date: February 20, 2013

Previous NaCl optimization focused on large general-purpose server and desktop CPUs; the "smallest" architecture targeted by previous NaCl optimization are ARMv7 CPUs with the NEON vector-instruction set [10]. Despite this focus on large processors, the NaCl designers claim in [4, Section 4] that

> "all of the cryptographic primitives in NaCl can fit onto much smaller CPUs: there are no requirements for large tables or complicated code"

This paper shows that this claim is actually correct.

The cryptographic primitives used by default in NaCl to provide public-key authenticated encryption are the Curve25519 elliptic-curve Diffie-Hellman key-exchange protocol [2], the Poly1305 authenticator [5], and the Salsa20 stream cipher [3]. The designers of NaCl announced, that the next release of NaCl will use the Ed25519 elliptic-curve signature scheme [7,8] to provide cryptographic signatures. This signature scheme—as described in the original paper and as implemented in this paper—uses the SHA-512 hash function [28].

We will put all software described in this paper into the public domain to maximize reusability of our results. We will furthermore discuss possibilities for public benchmarking with the editors of eBACS [9] and XBX [35]. Currently eBACS does not support benchmarking on AVR microcontrollers; XBX only supports benchmarking of hash functions.

Main contribution. There exists an extensive literature describing implementations of cryptographic *primitives* on AVR microcontrollers and other embedded processors. Some of them have been integrated into libraries that offer a set of cryptographic functionalities, e.g., AVR-Crypto-Lib [15], TinyECC [24], NanoECC [32], or the AVR Cryptolibrary from Efton s.r.o. [13]. These libraries are specifically tailored to match the specific restricted environment of the AVR.

This paper is the first that ports the entire NaCl library to AVR microcontrollers. These include the cryptographic primitives of Salsa20 [3], Poly1305 [5], Curve25519 [2], and Ed25519 [8]. All primitives are based—in contrast to existing AVR libraries—on at least 128-bit security and provide new speed records for that level of security. In addition, all functions run in constant time and do not contain secret-data-dependent branch conditions. This is important to provide a certain level of security against basic implementation attacks [22,25]. In particular the implementation is protected against *remote* side-channel attacks. Other cryptographic libraries for AVR do not concern about this issue. Moreover, the entire library is very small in size and requires only 17 351 bytes of code, no static RAM, and less than 1 479 bytes of stack memory; it therefore fits into very resource-constrained devices such as the very small ATmega family of microcontrollers, e.g., the ATmega32, ATmega328, and ATmega324A. Last but not least, we present new speed records for Salsa20 on AVRs and give first results of scalar multiplication for Curve25519 and signing and verifying using Ed25519 on AVR.

Roadmap. The paper is organized as follows. In Section 2, we briefly describe the AVR family of microcontrollers. Section 3 describes the NaCl library and

the general approach to porting it to AVR. In Section 4, we describe the implementation of Salsa20. In Section 5, we describe the implementation of Poly1305. Section 6 presents the implementations of Curve25519 and Ed25519 (including SHA-512). Results, a comparison with previous work, and a discussion are given in Section 7.

2 The 8-bit Family of AVR Microcontrollers

Atmel offers a wide range of 8-bit microcontrollers that can be mainly separated into three groups. High-end devices with high performance (ATxmega), mid-end devices featuring most functionality needed for the majority of applications (ATmega), and low-end devices with limited memory and processing power (ATtiny). Typical use cases of those devices are embedded systems such as motor control, sensor nodes, smart cards, networking, metering, medical applications, etc.

All those devices process data on 8-bit words. There are 32 general purpose registers available, R0-R31, which can be freely used by implementations. Some of them have special features like R26-R31, which are register pairs used to address 16-bit addresses in SRAM, i.e., X (R27:R26), Y (R29:R28), and Z (R31:R30). Some of those registers (R0-R15) can also only be accessed by a limited set of instructions (in fact only those who do not provide an immediate value as one operand).

The instruction set offers up to 90 instructions which are equal for all AVR devices. For devices with more memory or enhanced cores, it is extended by more than 30 additional instructions. The most important instruction for (public-key) cryptography is multiplication. It is not available for minimal cores such as the ATtiny or AT90Sxxxx family. But for enhanced cores like most of the ATmega and also all ATxmega cores, it allows (signed or unsigned) multiplication of two 8-bit words within two clock cycles. The 16-bit result of the multiplication is always stored in the register pair R1:R0. The software described in this paper makes use of these multipliers and does therefore not support the low-end ATtiny and AT90Sxxxx devices.

ATmega example configurations. We perform all benchmarks on an ATmega2560 which has a maximal clock frequency of 16 MHz, a flash storage of 256 KB and 8 KB of RAM. Other typical configurations of ATmega microcontrollers are, for example, the ATmega128 with a maximal clock frequency of 16 MHz, 128 KB of flash storage and 4 KB of RAM and the ATmega328 with a maximal clock frequency of 20 MHz, 32 KB of flash storage and 2 KB of RAM.

Radix-2^8 representation. The typical representation of integers of size larger than 8 bits on an 8-bit architecture is to split integers into byte arrays using radix 2^8. In other words, an m-bit integer x is represented as $n = \lceil m/8 \rceil$ bytes $(x_0, x_1, \ldots, x_{n-1})$ such that $x = \sum_{i=0}^{n} x_i 2^{8*i}$. We use this representation for all integers and elements of finite fields.

3 The NaCl Library

The Networking and Cryptography library (short: NaCl; pronounced: "salt") is a cryptographic library for securing Internet communication [4]. It was developed as one deliverable of the project CACE (Computer Aided Cryptography Engineering) funded by the European Commission. After CACE ended in December 2010, development of NaCl continued within the VAMPIRE virtual lab [23] of the European Network of Excellence in Cryptology, ECRYPT II [12]. The main features of the library are the following:

Easy usability. The library provides a high-level API for public-key authenticated encryption through one function call to `crypto_box`. The receiver of the message verifies the authentication and recovers the message through one function call to `crypto_box_open`. A pair of a public and a private key is generated through `cryto_box_keypair`. A similarly easy-to-use API is offered for cryptographic signatures: A function call to `crypto_sign` signs a message, `crypto_sign_open` verifies the signature and recovers the message, `crypto_sign_keypair` generates a keypair for use with this signature scheme. Implementors of information-security systems obtain high-security cryptographic protection without having to bother with the details of the underlying primitives and parameters. Those are chosen by the NaCl designers.

High security. The key sizes are chosen such that the security level of the primitives is at least 128 bits. Furthermore, NaCl is the only cryptographic library that systematically protects against timing attacks by avoiding loads from addresses that depend on secret data and avoiding branch conditions that depend on secret data. For further security features of NaCl see the extensive discussion in [4, Section 3].

High speed. The cryptographic primitives chosen for NaCl allow very fast implementations on a large variety of architectures.

No usage restrictions. The library is free of copyright restrictions. It is in the public domain. Furthermore the library avoids all patents that the authors are aware of. NaCl is free for download at `http://nacl.cr.yp.to/`.

3.1 Porting NaCl to AVRs

Reusing code. Porting a whole cryptographic library to a memory-restricted and storage-restricted environment such as AVR microcontrollers is different from porting each primitive in the library separately. To minimize code size we can use some functionalities (such as big-integer arithmetic) in multiple primitives. Sometimes this requires optimizing algorithm choices *across primitives*. For example, the Poly1305 authenticator described in Section 5 needs multiplication of 130-bit numbers; the Curve25519 key-exchange and Ed25519 signatures described in Section 6 need fast multiplication of 256-bit (or at least 255-bit) numbers. With the Karatsuba technique [20] we decompose the 256-bit (32×32-byte) multiplication into two 16×16-byte multiplications and one 17×17-byte

multiplication. The latter one can directly be used for the Poly1305 authenticator.

Secret load addresses. On all architectures targeted in previous NaCl optimization, loading data from an address that depends on secret data causes timing variation that can be used by an attacker to mount a timing attack. The reason is that memory access on all these architectures uses a hierarchy of transparent caches; the time required for a load operation depends on whether the requested data is in cache (*cache hit*) or not (*cache miss*). Memory access on the AVR microcontroller is not cached, it takes a constant amount of time. Loading data from a secret position on an AVR will not leak timing information. Avoiding loads from secret positions incurs performance penalties, we therefore decided to *not* avoid loads from secret addresses on the AVR.

Secret branch conditions. Conditional branches are an even more obvious source for timing variation than data loads. Even if both possible branches take the same amount of time to execute, branch conditions that depend on secret data will leak timing information on most architectures. The reason is that most processors use branch-prediction techniques to avoid pipeline stalls. If a branch is predicted correctly, the branch will incur only a small or no penalty; a mispredicted branch typically takes much more time.

AVR microcontrollers do not use any branch-prediction techniques so in principle one can write software that *does* use secret branch conditions and still runs in constant time. However, it is very tedious to review such code for constant-time behaviour and the performance benefits are relatively small. We therefore follow the strategy of all other NaCl optimizations and avoid all data flow from secret data to branch conditions.

Randomness generation. NaCl uses the operating-system's random-number generator and reads random bytes from `/dev/urandom` (see [4, Section 3, "Centralizing randomness"]). This is not possible on the AVR microcontroller. Our implementation of NaCl does not contain any cryptographically secure randomness generator. To test the key-generation functions that require randomness we used the deterministic `randombytes` function from the `try-anything` program of the SUPERCOP benchmarking suite. There are two different ways to address randomness generation on the AVR: One can use NaCl in a way that does not require randomness by computing key pairs on an external device and transferring them to the AVR. In NaCl, all operations except key-generation are deterministic. See [4, Section 3, "Avoiding unnecessary randomness"].

If one needs to generate keys on an AVR microcontroller it is necessary to include cryptographically secure randomness generation. One possible source of randomness is, for example, the jitter of the RC oscillator as described in [18].

Message lengths. In the C interface of NaCl, message lengths are passed as 64-bit unsigned integers (datatype `unsigned long long`). Adresses on the AVR ATmega microcontrollers have only 16 bits; we therefore omit expensive arithmetic on 64-bit integers to support messages of a length that would anyway not fit into the addressable memory.

Benchmarking. The cycle-count numbers of the various primitives presented in this paper have been obtained as follows. The numbers given in the following sections are the results of cycle-accurate simulations for an ATmega2560 microcontroller. The results given in the Section 7 (the results given in Table 1 in particular), are obtained through actual measurements on the same targeted microcontroller. For this purpose we re-implemented the 64-bit resolution `cpucycles` cycle counter included in NaCl and the eBACS benchmarking suite SUPERCOP [9] for AVR. We combine the 8-bit and the 16-bit cycle counters into one 24-bit cycle counter and increase the overall count by 2^{24} for an overflow interrupt of the higher counter. The cycle counts include an 247-cycle overhead (284-cycle overhead for the low-area variant) for function call and reading the 64-bit cycle count; it is reported as "empty" benchmark in Table 1. We measured this overhead by subsequently calling an empty function and reading the cycle counter many times and computing the differences of the measurements. We also measured the overhead for reading the cycle counter without the overhead of function calls by computing differences of subsequent readings of the cycle counter. This overhead is 230 cycles (274 cycles for the low-area variant); it is reported as "nothing" benchmark in Table 1.

4 Implementation of Salsa20

Salsa20 is a stream cipher which has been proposed in 2005 [3]. It has been included in the final portfolio of the eSTREAM project initiated by the European Network of Excellence for Cryptology (ECRYPT) in 2004. The cipher consists of 20 rounds[3] where an internal state is modified by various (logical and arithmetic) transformations. To encrypt a message, a 32-byte key is used.

4.1 High-speed implementation

The Salsa20 stream cipher is realized in the library function `crypto_stream`. After calling the function, a message with variable length is split into several input blocks with 64 bytes in length. The cipher is then applied on each message block by calling the function `crypto_core`. This function first initializes a 32-byte state and starts the round calculation afterwards. Both functions have been implemented in assembly to improve the performance of Salsa20.

Initialization of the State. The function `init_core` mainly consists of 7 loop iterations where the state x (and a copy of the state j which is later added to the cipher output) gets initialized with the 32-byte key, the 64-byte input, and a 16-byte nonce. The initialization takes 718 clock cycles in total.

[3] Note that there also exist round-reduced versions of Salsa20, e.g., Salsa20/12 applying 12 rounds instead of 20.

Round Calculation. The round-calculation function provides the most promising potential to increase the speed of Salsa20. It consists of ten loop iterations that include 8 quarterround function calls (thus 80 function calls in total). Within one quarterround function, three different 32-bit operations (addition, bitwise addition, and rotations) are performed on either the rows or the columns of the state x.

We implemented the following optimizations. First, we re-used all 32 available registers of the AVR to avoid unnecessary storing and loading from the stack which is costly in terms of memory and speed. For this, we passed the addresses of the current row or column of the state in the registers R18-R25. The values of the state are then loaded into the registers R0-R15. The register pair R17:R16 is reserved to store the 16-bit base address. It will not be modified within the quarterround function. The remaining address registers R26-R31 are used for fast addressing during the round transformations. They allow to implicitly decrement the addresses before or after a `ST` (store) or `LD` (load) instruction. Second, the state variables are modified in-place. This means that the state is directly modified without needing extra variables and copy instructions. Third, the shift operations by 7 and 9 have been realized by cheap logical shift (`LSR` and `LSL`) and rotate through carry instructions (`ROR` and `ROL`). Shifting by 13 and 18 has been realized using the `MUL` instruction by multiplying with the constants $2^5 = 32$ and $2^2 = 4$.

One quarterround function call requires 174 clock cycles in total. The entire round calculation needs 15 623 clock cycles. The entire `crypto_stream` function needs 18 166 clock cycles for a 64-byte message. The code size of Salsa20 is 1 750 bytes.

4.2 Low-area implementation

For the low-area version, we looped the final addition of j at the end of the quarterround function. The remaining assembly parts are already optimized in terms of low area. We also used the `-Os` compiler flag to optimize for small code size. With these modifications, the performance is slightly reduced by 697 clock cycles, resulting in 18 863 clock cycles for `crypto_stream`; the code size is reduced by 658 bytes to only 1 092 bytes, i.e., 37.6 %.

5 Implementation of Poly1305

Poly1305 is a message authentication code (MAC) proposed in 2005 [5]. The name is related to the used underlying polynomial $2^{130} - 5$. A message m with variable size n is authenticated using a (random) 32-byte one-time secret key s (and a 16-byte nonce). The secret key s consists of two parts each 16-bytes in length, i.e., $s = (k, r)$. First, the message m is split into 16-byte blocks where each block is padded with a 1. The resulting 17-byte chunks c_i, where $i \in [1, q]$ and $q = \lceil n/16 \rceil$, are then represented as unsigned litte-endian integer. After

that, one addition and one modular multiplication is performed for each chunk c resulting in the 16-byte authenticator h, i.e.,

$$h = (((c_1 \cdot r^q + c_2 \cdot r^{q-1} + ... + c_q \cdot r^1) \bmod 2^{130} - 5) + s) \bmod 2^{128}. \quad (1)$$

5.1 High-speed implementation

The most time-consuming operation in Poly1305 is modular multiplication in the field $2^{130} - 5$. In order to obtain high speeds, we implemented both multiplication and reduction in assembly. To save code size, we implemented a 2^{136}-bit multiplier that is also (re)used by the Karatsuba-multiplier implementation of Curve25519 and Ed25519 as described in Section 6.

Multiplication of $2^{136} \times 2^{136}$. There exist various ways to implement large-integer multiplication, for example, the widely used schoolbook or Comba multiplication. On AVRs, it has been shown by various papers that a combination of both techniques significantly helps in speeding up the computation; cf., [16,24,32,34].

We followed a similar approach by breaking the 136-bit multiplication into 8×8-byte, 9×9-byte, and 9×8-byte multiplications and combine the partial results within each block in a conventional schoolbook approach. The 17×17-byte multiplication takes 1 967 cycles. The code size of the fully unrolled implementation is 2 944 bytes.

Reduction mod $2^{130} - 5$ on AVR. Modular reduction has been implemented as follows. Since the prime $p = 2^{130} - 5$ is a Mersenne-like prime, we can apply fast reduction by using simple shifts and additions only which are relatively cheap on AVRs. Consider the integer $x \in [0, p^2)$ and let $x = x_1 \cdot 2^{130} + x_0$ be the result of the multiplication. Then, we can exploit the congruence $2^{130} \equiv 5$ and we can add the shifted higher part $x_1 \cdot 5$ to the lower part x_0, i.e., $x_1 \cdot 5 + x_0 = x_1 + (x_1 \ll 2) + x_0$. After this operation, the result might be still larger than p, thus another reduction iteration has to be performed on the lower words using the same technique.

We can optimize the operation by exploiting the gap between 2^{128} and $2^{130} - 5$ on the AVR. Since we operate on radix-2^8, the integer x is represented as $x = x_1 \cdot 2^{128} + x_0$ where x_0 and x_1 are 16-byte arrays, i.e., $x_j = \sum_{i=0}^{15} x_i 2^{8i}$ with $j = \{0, 1\}$. If we simply add the higher part x_1 to the lower part x_0, we have an implicit left shift of x_1 by 2. We then only have to shift x_1 two bits to the right to get the reduced result $x \equiv x_1 + (x_1 \gg 2) + x_0 \bmod p$. A right-shift operation by 2 can be done within 4 clock cycles on AVRs by repeating the following two instructions: a Logical Shift Right (`LSR`) instruction (which shifts the LSB to the carry register) and a Rotate Right Through Carry (`ROR`) instruction which rotates a byte by shifting the carry into the MSB. Note also that the first two LSBs of x_1 have to be set to zero before adding it with x_0 in order to get the correct result and to eliminate the two MSBs of the last word of x_0.

5.2 Low-area implementation

For the low-area version of Poly1305, we implemented three operations in a loop, i.e., two initializations of intermediate variables and the addition operation. For the latter operation we simply re-used the function `bigint_add`, which is also used for scalar arithmetic in Ed25519. These modifications have only a slight impact in performance (15 058 clock cycles are needed for a 64-byte message instead of 14 312) but the code size is reduced from 946 bytes to only 466, i.e., 50.7 % code-size reduction.

6 Curve25519 and Ed25519

In 2006, Bernstein introduced the Curve25519 elliptic-curve Diffie-Hellman key-exchange primitive and the corresponding high-speed software for various x86 CPUs [2]. Curve25519 uses the elliptic curve defined by the equation $E : y^2 = x^3 + 486662x^2 + x$ over the field $\mathbb{F}_{2^{255}-19}$. The scalar multiplication performed in Curve25519 uses the x-coordinate-based differential addition introduced by Montgomery in [27, Section 10]. The main computational effort for the scalar multiplication are 255 so called ladder steps, 255 conditional swaps, each based on one bit of the scalar, and one inversion in $2^{2^{255}-19}$. Each of the laddersteps consists of 5 multiplications, 4 squarings, 1 multiplication with the constant 121666, 4 additions, and 4 subtractions in $\mathbb{F}_{2^{255}-19}$.

In 2011, Bernstein, Lange, Duif, Schwabe and Yang introduced the Ed25519 elliptic-curve digital-signature scheme and presented corresponding high-speed software for Intel Nehalem/Westmere processors [7,8]. The signatures are based on arithmetic on the twisted Edwards curve [6] defined by the equation $E : x^2 + y^2 = 1 - \frac{121665}{121666}x^2y^2$ over $\mathbb{F}_{2^{255}-19}$. This curve is birationally equivalent to the Montgomery curve used in the Curve25519 key-exchange software. The main computational effort for Ed25519 key-pair generation and signing is one fixed-base-point scalar multiplication with a secret scalar. The main computational effort for signature verification is one point decompression (Ed25519 stores only the y coordinate and one bit of the x coordinate of public keys) and one double-point scalar multiplication with public scalars. One of the two points involved in this double-point scalar multiplication is the fixed-base-point also used in key-pair generation and signing.

6.1 High-speed implementation

Arithmetic in $\mathbb{F}_{2^{255}-19}$. The computations of both Curve25519 key exchange and Ed25519 signatures break down to operations in the field $\mathbb{F}_{2^{255}-19}$. The most speed-critical operations are multiplications and squarings. We decided to not specialize squarings to save code size.

Multiplication is implemented as one level of Karatsuba multiplication, that breaks the 32×32-byte multiplication into two 16×16-byte multiplications and one 17×17-byte multiplication. Note that the latter multiplication is also used for

the Poly1305 authenticator described in Section 5. Next to the multiplications, two 16-byte additions, two 33-byte additions, and two 33-byte subtractions are required in addition to accumulate the intermediate results. The entire 32×32-byte Karatsuba multiplication takes 6 868 cycles; this is slightly slower than the current state of the art presented at CHES 2011 [19] (6 208 cycles); but we save in code size, especially for the low-area variant as described later. For the completely unrolled high-speed version of the 32×32-byte multiplication, 7 184 bytes of code are required.

Throughout the whole computation we do not reduce modulo $2^{255} - 19$, but instead only modulo $2^{256}-38$. Only at the very end we "freeze" the values modulo $2^{255} - 19$. To perform modular reduction after a multiplication or squaring we multiply the upper 32 bytes of the 64-byte result by 38 and then add those to the lower 32 bytes. This will leave us with a 33-bit value. We multiply the highest bit again by 38 and add the 2-byte result to the lowest two bytes and ripple the carry through all 32 bytes. This may again produce a carry which we multiply by 38 and add to the lowest byte. Note that this final addition can not produce a carry. After an addition or subtraction we simply multiply the final carry bit by 38 and add to (or subtract from) the lowest byte; then ripple through the carry and again multiply the carry by 38 and add to the lowest byte. These reductions after multiplication and addition use fully unrolled loops. We use a separate function call to the modular reduction after multiplication and squaring. This way we are able to reuse the 32×32-byte multiplication for arithmetic on scalars in Ed25519 signature verification. Addition and subtraction in $\mathbb{F}_{2^{255}-19}$ do not use separate function calls to reduction. They have been also fully unrolled.

Curve25519. Our Curve25519 software uses the same sequence of 255 Montgomery ladder steps and 255 conditional swaps as previous optimized implementations of Curve25519 [10,2]. The conditional swaps neither use lookups from secret addresses nor (as previously explained) secret branch conditions; a conditional swap between two values a and b depending on one secret bit s is computed as two conditional moves; each conditional move is computed by first expanding the secret bit s to an all-one or all-zero mask $\overline{s}$ and then computing $a \leftarrow a$ XOR ($\overline{s}$ AND (a XOR b)).

The final inversion in $\mathbb{F}_{2^{255}-19}$ is computed as exponentiation with $2^{255} - 20$ using the same sequence of 254 squarings and 11 multiplications as [2]. We implemented this sequence of function calls in C and used the compiler flags `-mmcu=atmega2560 -Os -mcall-prologues` to translate it.

Ed25519 key-pair generation and signing. The fixed-base-point scalar multiplication in key-pair generation and signing is implemented through a signed-fixed-window scalar multiplication with window size 4. The elliptic-curve arithmetic uses the extended coordinates introduced in [17]. In total the fixed-base-point scalar multiplication requires 64 table lookups, 63 additions of a precomputed multiple of the basepoint to a point in extended coordinates, and 252 doublings in extended coordinates. At the end of this computation we need one inversion and two multiplications $\mathbb{F}_{2^{255}-19}$ to convert to affine coordinates. The precomputed multiples of the base point are in an array marked as `PROGMEM`.

This way they do not occupy space in the data segment in RAM but only in the (much larger) flash memory. Before performing the fixed-base-point scalar multiplication we copy this table of precomputed points into a space on the stack to avoid (secretly indexed) lookups from flash memory.

Ed25519 verification. We perform point decompression of the public key in the same way as explained in [8, Section 5]. We implement the required exponentiation by $2^{252} - 3$ the same way as the inversion: A sequence of function calls to multiplications and squarings implemented in C and compiled with the flags `-mmcu=atmega2560 -Os -mcall-prologues`.

For the double-scalar multiplication we use Straus' algorithm [31] with window-size 1, a special case that is sometimes referred to as "Shamir's trick". For the multiplication of 256-bit scalars modulo the group order we use the 32×32-byte multiplication and subsequent Barrett reduction [1].

SHA-512. Ed25519 signatures need a 512-bit-output hash function; the original paper [8] uses SHA-512 but the authors comment that they "will not hesitate to recommend Ed25519-SHA-3 after SHA-3 is standardized". In order to provide a compatible implementation to the Ed25519 implementations currently included in SUPERCOP [9] we also use Ed25519-SHA-512. We implemented all speed-critical low-level functions, in particular arithmetic on 64-bit integers, in assembly. This assembly implementation unrolls all length-8 loops. Calls to the low-level assembly functionalities are implemented in C. Compiling this SHA-512 C code with the `-O3` flag, which we use for most files in the high-speed version, results in unacceptably large code; for SHA-512 we therefore use compiler flags `-mmcu=atmega2560 -Os -mcall-prologues`.

6.2 Low-area implementation

Arithmetic in $\mathbb{F}_{2^{255}-19}$. The main difference in the implementation of finite-field arithmetic for the low-area implementation is that we get rid of the 16×16-byte multiplication. Instead we copy the arguments to 17-byte arrays with leading zero byte and use the 17×17-byte multiplication. The resulting assembly implementation of 32×32-byte multiplication that performs 3 calls to 17×17 byte multiplication and all necessary additions and copies for the Karatsuba multiplication has a size of 3 358 bytes (53.25 % less code size compared to the high-speed version). The runtime is increased to 8 322 clock cycles.

Aside from that change we do not unroll the loops in the modular reduction after multiplication, addition, and subtraction to further reduce code size.

Curve25519. The high-level implementation of Curve25519 is the same for the small-area implementation as for the high-speed implementation.

Ed25519 key-pair generation and signing. For the fixed-base-point scalar multiplication we also use a signed-fixed-window scalar-multiplication algorithm. Instead of window size 4 (as in the high-speed implementation) we use a window size of only 2 to save space in flash and RAM.

Ed25519 verification. The high-level implementation of key-pair generation and signing is the same for the small-area implementation as for the high-speed implementation.

SHA-512. SHA-512 uses almost the same code as same in the high-speed implementation. The only difference is that we do not unroll the 3 length-8 loops in the σ-transformation of SHA-256. This change slightly shrinks the code size without significantly hurting performance.

7 Results

In this section we report benchmarks of our software and give a comparison with previous results. As described in Subsection 3.1, the benchmarks are not obtained in a simulator but by measuring cycles on an actual ATmega2560 microcontroller clocked at 16 MHz (on the Arduino Mega 2560 development board). Measuring cycles incurs a certain overhead; we give this overhead as a "nothing" benchmark, i.e. simply differences of subsequent readings to the cycle-counter. The reported numbers are the median of the cycle counts of 20 runs of the respective primitive.

We compiled all C software with avr-gcc version 4.7.0. For the high-speed implementation we used compiler flags `-mmcu=atmega2560 -O3` where not otherwise reported; for the low-area implementation we used the compiler flags `-mmcu=atmega2560 -Os -mcall-prologues`. Our implementation does not use any space in the data segment and no dynamic memory allocation; so RAM is only used by the stack[4]. We measured stack space by writing a canary value to the whole stack before running the actual function; then reading later how many of the canary bytes have been overwritten. Reporting code sizes for individual primitives does not make much sense because of large portions of code that is shared between the primitives (for example Curve25519 and Ed25519 share the code for field arithmetic in $\mathbb{F}_{2^{255}-19}$). Instead, we report the code size (i.e. required space in the flash memory) for both implementations of the whole library. These sizes were obtained with avr-size from GNU binutils version 2.20.1.20100303. Our results are summarized in Table 1.

Comparison with Related Work. To the authors knowledge, there exist three resources that present results of Salsa20 on AVR microcontrollers. Meiser et al. [26] and Eisenbarth et al. [14] reported results of Salsa20 implemented in C and assembly. Their fastest design needs 17 812 clock cycles for one 64-byte message block needing 2 984 bytes of code. Their low-area variant needs 18 400 clock cycles and 1 452 bytes of code. Both implementations need 280 bytes of RAM. There is also a C implementation of Salsa20 in the AVR-CryptoLib [15] written by Daniel Otte. His implementation requires 723 clock cycles for initializing the state and 94 476 clock cycles for encryption.

[4] We observed that earlier versions of avr-gcc, for example, avr-gcc 4.5, place some constants in the data segment; gcc-4.7 stores those constants in program memory.

Table 1. Benchmark results of NaCl on the AVR ATmega2560 microcontroller

Primitive		**Message bytes**	**Cycles**	**Stack bytes**
nothing	high-speed		230	
	low-area		274	
empty	high-speed		247	
	low-area		284	
Salsa20	high-speed	8	17 138	266
		64	18 166	
		576	159 510	
		1024	283 186	
		2048	565 873	
	low-area	8	17 830	275
		64	18 863	
		576	165 287	
		1024	293 408	
		2048	586 255	
Poly1305	high-speed	8	4 146	114
		64	14 312	
		576	122 152	
		1024	216 512	
		2048	432 191	
	low-area	8	4 509	114
		64	15 058	
		576	126 962	
		1024	224 878	
		2048	448 685	
SHA-512	high-speed	8	535 901	689
		64	535 733	
		576	2 655 521	
		1024	4 775 501	
		2048	9 015 172	
	low-area	8	607 082	669
		64	606 916	
		576	3 012 120	
		1024	5 417 516	
		2048	10 228 019	
Primitive		**Operation**	**Cycles**	**Stack bytes**
Curve25519	high-speed	`crypto_scalarmult_base`	22 954 658	681
		`crypto_scalarmult`	22 954 657	681
	low-area	`crypto_scalarmult_base`	28 043 134	919
		`crypto_scalarmult`	28 043 124	922
Ed25519	high-speed	`crypto_sign_keypair`	21 924 771	1 566
		`crypto_sign`	23 211 611	1 642
		`crypto_sign_open`	32 619 197	1 317
	low-area	`crypto_sign_keypair`	32 937 940	1 282
		`crypto_sign`	34 342 230	1 289
		`crypto_sign_open`	40 093 186	1 349

NaCl implementation	**Code size (in bytes)**
high-speed	28 883
low-area	17 373

In view of elliptic-curve implementations on AVR, there exist many results presented for example in [16,21,33,34]. Most of these results are hard to compare since the implementations differ in various ways such as in the size of the underlying finite field, the used ECC group formulae, the multiplication technique (both in terms of group and field arithmetic), and additionally implemented higher-level protocols (e.g., hash functions, signing and verifying of messages, random number generation, ...). For example, one of the first who reported the performance of ECC on an ATmega128 are Gura et al. [16] who presented their results at CHES 2004. They implemented ECC over the NIST standardized curves over prime-fields $\mathbb{F}_{p160}$, $\mathbb{F}_{p192}$, and $\mathbb{F}_{p224}$. Their implementation needs 17.52 million clock cycles for a single scalar multiplication on the curve over $\mathbb{F}_{p224}$. Uhsadel et al. [33] reported around 10 million cycles for a 160-bit scalar multiplication.

One of the few AVR libraries that support also higher-level protocols are TinyECC, NanoECC, or CRS-AVR010X-ECC. TinyECC has been presented by Liu et al. [24] in 2008. The library implements ECDSA, ECDH, and ECIES over the SECG curves over $\mathbb{F}_{p128}$, $\mathbb{F}_{p160}$[5], and $\mathbb{F}_{p192}$. A single scalar multiplication needs 6.48 million clock cycles. Signing using ECDSA needs 16 million clock cycles and 27 million cycles in addition to pre-compute the base-point multiples of the implemented sliding window scalar-multiplication method. The entire library needs between 15 492 and 19 308 bytes of code (depending on the used multiplication method) and around 1 500 bytes of RAM. The low-area variant needs 10 180 bytes of code and 152 bytes of RAM. NanoECC has been proposed by Szczechowiak et al. [32]. The library implements the NIST-K163 Koblitz curve over $\mathbb{F}_{p160}$. They reported 9.37 million clock cycles for one scalar multiplication and the code size of the library is 46 100 bytes[6] and the RAM usage is 1 800 bytes. There exist also another library called CRS-AVR010X-ECC [29] that implements ECDSA and ECDH on SECG curves over $\mathbb{F}_{p160}$, $\mathbb{F}_{p192}$, $\mathbb{F}_{p224}$, and $\mathbb{F}_{p256}$. The implementation on the curve over $\mathbb{F}_{p256}$ needs 5 to 8 kB of code and 750 to 900 bytes of RAM. Signing using ECDSA requires 76.8 million cycles. Their high-speed implementation requires only 27.2 million cycles with an additional memory of 16 384 bytes.

Recently, Chu et al. [11] set new speed records for a single scalar multiplication on Twisted Edwards curves on AVRs. Their implementation needs only 5.9 million clock cycles for a 160-bit curve on an ATmega128. However, the authors aimed for high-speed without considering implementation attacks, e.g., they implemented the conventional double-and-add method and used data-dependent branch conditions which can be exploited in implementation attacks [22,25].

Discussion. As explained in the introduction, our implementation of NaCl does not aim at highest speed at all costs. Instead we aimed at good speeds with a moderate RAM and ROM usage. With this paper we are hoping for feedback

[5] Curve *secp160r1* has been used in [24] for evaluating the performance of TinyECC.

[6] NanoECC is based on the MIRACLE (Multi-precision Integer and Rational Arithmetic C/C++ Library) [30], which provides many functions and tools to implement higher-level protocols.

from potential users of AVR NaCl telling us what the specific requirements of their application are. For applications that require higher speeds for a specific primitive there are various possibilities for speedups, in particular in Curve25519 and Ed25519:

- Arithmetic in $\mathbb{F}_{2^{255}-19}$ does not use special code for squarings but instead uses calls to the muliplication. A specialized squaring implementation would speed up both Curve25519 and Ed25519.
- The Karatsuba multiplier used for multiplication in $\mathbb{F}_{2^{255}-19}$ is only slightly slower than the operand-caching multiplication presented in [19]; however, switching to operand-caching multiplication would offer further speedups for Curve25519 and Ed25519.
- The multiplication with the small constant 121 666 in Curve25519 is not specialized; again we are using a call to the full multiplication. A specialized function for multiplication with this constant would speed up Curve25519.
- Ed25519 signature verification uses Straus' algorithm with window size 1 instead of, for example, a sliding-window algorithm that would require significantly more RAM. If RAM usage is not a critical limitation we could thus easily speed up signature verification.
- We do not expect users of AVR NaCl to have any use for the fast batch verification of signatures; processing *many* signatures in short time is not exactly the typical domain for embedded microcontrollers. If applications benefit from fast batch verification and are willing to spend some space in RAM, we could also include the fast batch verification based on the Bos-Coster multi-scalar-multiplication algorithm described in [8, Section 5].

References

1. Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology – CRYPTO '86*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer-Verlag Berlin Heidelberg, 1987. 11
2. Daniel Bernstein. Curve25519: New Diffe-Hellman Speed Records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *9th International Conference on Theory and Practice in Public-Key Cryptography - PKC 2006, New York, NY, USA, April 24-26, 2006. Proceedings*, volume 3958, pages 207–228, 2006. http://cr.yp.to/papers.html#curve25519. 2, 9, 10
3. Daniel Bernstein. *New Stream Cipher Designs*, volume 4986, chapter The Salsa20 family of stream ciphers, pages 84–97. Springer-Verlag Berlin, 2008. http://cr.yp.to/papers.html#salsafamily. 2, 6
4. Daniel Bernstein, Tanja Lange, and Peter Schwabe. The Security Impact of a New Cryptographic Library. In Alejandro Hevia and Gregory Neven, editors, *2nd International Conference on Cryptology and Information Security in Latin America, Santiago, Chile, October 7-10, 2012. Proceedings*, volume 7533 of *Lecture Notes in Computer Science*, pages 159–176. Springer Berlin Heidelberg, 2012. http://cryptojedi.org/papers/#coolnacl. 1, 2, 4, 5

5. Daniel J. Bernstein. The Poly1305-AES Message-Authentication Code. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers*, volume 3557 of *Lecture Notes in Computer Science*, pages 32–49. Springer, February 2005. http://cr.yp.to/papers.html#poly1305. 2, 7
6. Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. Twisted edwards curves. In Serge Vaudenay, editor, *Progress in Cryptology – AFRICACRYPT 2008*, volume 5023 of *Lecture Notes in Computer Science*, pages 389–405. Springer-Verlag Berlin Heidelberg, 2008. http://cr.yp.to/papers.html#twisted. 9
7. Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 124–142. Springer-Verlag Berlin Heidelberg, 2011. see also full version [8]. 2, 9, 16
8. Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012. http://cryptojedi.org/papers/#ed25519, see also short version [7]. 2, 9, 11, 15, 16
9. Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT benchmarking of cryptographic systems. http://bench.cr.yp.to (accessed 2013-01-31). 2, 6, 11
10. Daniel J. Bernstein and Peter Schwabe. Neon crypto. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES 2012*, volume 7428 of *Lecture Notes in Computer Science*, pages 320–339. Springer Berlin Heidelberg, 2012. 2, 10
11. Dalin Chu, Johann Großschädl, and Zhe Liu. Twisted Edwards-Form Elliptic Curve Cryptography for 8-bit AVR-based Sensor Nodes. In *Cryptology ePrint Archive: Report 2012/730*, 2012. 14
12. European network of excellence in cryptology ii. http://www.ecrypt.eu.org/index.html (accessed 2013-01-18). 4
13. Efton. 8051 and AVR Cryptolibrary. Available online at http://www.efton.sk/crypt/index.htm. 2
14. Thomas Eisenbarth, Sandeep Kumar, Christof Paar, Axel Poschmann, and Leif Uhsadel. A Survey of Lightweight-Cryptography Implementations. *IEEE Design & Test of Computers - Design and Test of ICs for Secure Embedded Computing*, 24(6):522–533, November-December 2007. ISSN 0740-7475. 12
15. Das Labor e.V. AVR-Crypto-Lib. Available online at http://www.das-labor.org/wiki/AVR-Crypto-Lib/en. 2, 12
16. Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. Comparing Elliptic Curve Cryptography and RSA on 8-Bit CPUs. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems – CHES 2004, 6th International Workshop, Cambridge, MA, USA, August 11-13, 2004, Proceedings*, volume 3156 of *Lecture Notes in Computer Science*, pages 119–132. Springer, 2004. 8, 14
17. Huseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, , and Ed Dawson. Twisted edwards curves revisited. In Josef Pieprzyk, editor, *Advances in Cryptology – ASIACRYPT 2008*, volume 5350 of *Lecture Notes in Computer Science*, pages 326–343. Springer-Verlag Berlin Heidelberg, 2008. http://eprint.iacr.org/2008/522/. 10
18. Josef Hlaváč, Róbert Lórencz, and Martin Hadáček. True random number generation on an Atmel AVR microcontroller. In *2010 2nd International Conference on*

Computer Engineering and Technology (ICCET), volume 2, pages 493–495. IEEE, 2010. 5

19. Michael Hutter and Erich Wenger. Fast multi-precision multiplication for public-key cryptography on embedded microprocessors. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011, 13th International Workshop, Nara, Japan, September 28 – October 1, 2011. Proceedings*, volume 6917 of *Lecture Notes in Computer Science*, pages 459–474. Springer-Verlag Berlin Heidelberg, 2011. 10, 15
20. A. Karatsuba and Yu. Ofman. Multiplication of Multidigit Numbers on Automata. *Soviet Physics-Doklady*, pages 595–596, 1963. 4
21. Anton Kargl, Stefan Pyka, and Hermann Seuschek. Fast Arithmetic on ATmega128 for Elliptic Curve Cryptography. Cryptology ePrint Archive (http://eprint.iacr.org/), Report 2008/442, October 2008. 14
22. Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*, pages 104–113, 1996. 2, 14
23. Tanja Lange. Vampire – virtual applications and implementations research lab, 2007. http://hyperelliptic.org/ECRYPTII/vampire/ (accessed 2013-01-28). 4
24. An Liu and Peng Ning. TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks. In *International Conference on Information Processing in Sensor Networks - IPSN 2008, April 22-24, 2008, St. Louis, Missouri, USA, Proceedings.*, pages 245–256, St. Louis, MO, April 2008. 2, 8, 14
25. Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks – Revealing the Secrets of Smart Cards*. Springer, 2007. ISBN 978-0-387-30857-9. 2, 14
26. Gordon Meiser, Thomas Eisenbarth, Kerstin Lemke-Rust, and Christof Paar. Efficient Implementation of eSTREAM Ciphers on 8-bit AVR Microcontrollers. In *Industrial Embedded Systems, 2008. SIES 2008. International Symposium on*, pages 58 –66, june 2008. 12
27. Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987. http://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866113-7/S0025-5718-1987-0866113-7.pdf. 9
28. National Institute of Standards and Technology (NIST). FIPS-180-3: Secure Hash Standard, October 2008. Available online at http://www.itl.nist.gov/fipspubs/. 2
29. Center of Mathematical Modeling Sigma. CRS-AVR010X-ECC. Available online at http://www.cmmsigma.eu/products/crypto/crs_avr010x.en.html. 14
30. Michael Scott. MIRACLE – A Multiprecision Integer and Rational Arithmetic C/C++ Library. Available online at http://www.shamus.ie, 2003. 14
31. Ernst G. Straus. Addition chains of vectors (problem 5125). *American Mathematical Monthly*, 70:806–808, 1964. http://cr.yp.to/bib/1964/straus.html. 11
32. Piotr Szczechowiak, Leonardo B. Oliveira, Michael Scott, Martin Collier, and Ricardo Dahab. NanoECC: Testing the Limits of Elliptic Curve Cryptography in Sensor Networks. In Roberto Verdone, editor, *Wireless Sensor Networks 5th European Conference, EWSN 2008, Bologna, Italy, January 30-February 1, 2008. Proceedings.*, volume 4913 of *Lecture Notes in Computer Science*, pages 305–320. Springer, 2008. 2, 8, 14
33. Leif Uhsadel, Axel Poschmann, and Christof Paar. Enabling Full-Size Public-Key Algorithms on 8-bit Sensor Nodes. In Frank Stajano, Catherine Meadows, Srdjan Capkun, and Tyler Moore, editors, *Security and Privacy in Ad-hoc and Sensor*

Networks 4th European Workshop, ESAS 2007, Cambridge, UK, July 2-3, 2007. Proceedings., volume 4572 of *Lecture Notes in Computer Science*, pages 73–86. Springer, 2007. 14

34. Haodong Wang and Qun Li. Efficient Implementation of Public Key Cryptosystems on Mote Sensors. In *Information and Communications Security 8th International Conference, ICICS 2006, Raleigh, NC, USA, December 4-7, 2006. Proceedings*, volume 4307 of *Lecture Notes in Computer Science*, pages 519–528. Springer, 2006. 8, 14
35. Christian Wenzel-Benner and Jens Gräf. XBX: eXternal Benchmarking eXtension . http://xbx.das-labor.org/trac/wiki/WikiStart (accessed 2013-01-31). 2