IEICE TRANS. INF. & SYST., VOL.E99–D, NO.8 AUGUST 2016

PAPER Special Section on Security, Privacy and Anonymity of Internet of Things

Implementation of μNaCl on 32-bit ARM Cortex-M0

Toshifumi NISHINAGA[†], *Nonmember and* **Masahiro MAMBO**[†], *Member*

SUMMARY By the deployment of Internet of Things, embedded systems using microcontroller are nowadays under threats through the network and incorporating security measure to the systems is highly required. Unfortunately, microcontrollers are not so powerful enough to execute standard security programs and need light-weight, high-speed and secure cryptographic libraries. In this paper, we port NaCl cryptographic library to ARM Cortex-M0(M0+) Microcontroller, where we put much effort in fast and secure implementation. Through the evaluation we show that the implementation achieves about 3 times faster than AVR NaCl result and reduce half of the code size.

key words: *Cortex-M0, crypto library, embedded, microcontroller, IoT*

1. Introduction

Nowadays Internet of Things(IoT) is one of the most emerging technologies by the spread of the smartphone and low-price wireless module. Some types of IoT are composed of embedded systems using microcontroller. Such systems are connected through the Internet and it is unavoidable to take some security measure against threats through the network. Unfortunately, microcontrollers are not so powerful enough to execute standard security programs and need light-weight, high-speed and secure cryptographic libraries.

The microcontroller is one-chip controller that contains CPU, RAM, program memory and peripherals. It does not have enough power, but the cost is quite cheaper than the personal computer. It has been embedded in product that performs a simple control. The microcontroller's RAM and program memory are often have a very small size, e.g. the general Cortex-M0(M0+) memory size is about 8-KB to 128-KB. Therefore, the program used in microcontroller is required to have a small code size and a small amount of RAM. If the microcontroller has large memory, the maker can implement more application to one microcontroller in order to reduce the cost. Therefore, the cryptographic library is expected to have a small size. In addition, the program is required to achieve high-speed for energy-saving and shorten the response time for efficiency. A typical microcontroller has 8, 16 or 32-bit length architecture and one of these architectures is selected depending on the purpose. The bit length of the architecture is related to data length that can be processed at a time. For example, 32-bit microcontroller can process a 32-bit value at a time. Such a microcontroller is 4 times faster than 8-bit microcontroller. Therefore, it is preferable to use high-speed 32-bit microcontrollers in order to manage the process for the networking and security.

ARM Cortex-M0(and M0+) microcontroller is one of the 32-bit microcontrollers that have characteristics of low-cost and power-saving. They are expected to be used for the communication of the sensor nodes. In addition, the advent of mbed [1] platform will allow high-speed prototyping more easily by using ARM microcontroller. ARM microcontrollers will presumably continue to deploy further in the world. Therefore, we should support mbed platform at the time of make programs, in order to spread the programs for ARM microcontroller.

The Networking and Cryptography library NaCl pronounced "salt" [2] is a cryptographic library for securing Internet communication, developed by Daniel J Bernstein. This library has been designed for easy-usability, high-security and high-speed. NaCl has been re-designed as μNaCl for the microcontroller, and developed to AVR NaCl [3] for the AVR microcontroller. Unfortunately, AVR microcontrollers are not so powerful enough for encryption and communication for sensors.

Later, M0NaCl [4] was developed, that is μNaCl implementation for ARM Cortex-M0 microcontroller. However, only high-speed Curve25519 was implemented, and full function of NaCl cannot be used.

In this paper, we port full function of NaCl cryptographic libraries to ARM Cortex-M0(M0+) microcontroller, where we put much effort in fast and secure implementation, and evaluate the implementation.

This paper is organized as follows. In Sect. 2, we explain the NaCl and ARM Cortex-M0(M0+) microcontroller. Section 3 describes the implementation of NaCl. Results, comparison with previous work and discussion are given in Sect. 5.

2. Background

2.1 Networking and Cryptography Library(NaCl) and μNaCl

The Networking and Cryptography library NaCl pronounced "salt" [2] is a cryptographic library for securing Internet communication, developed by Daniel J Bernstein. This library has been designed for easy-usability, high-security and high-speed.

Manuscript received November 30, 2015.
Manuscript revised April 15, 2016.
Manuscript publicized May 31, 2016.
[†]The authors are with Kanazawa University, Kanazawa-shi, 920–1192 Japan.
DOI: 10.1587/transinf.2015INP0013

Table 1 NaCl Primitives

DH-Key exchange protocol	Curve25519 [5]
Stream Cipher	Salsa20 [6]
Public-key Digital Signature	Ed25519
MAC	Poly1305 [7]
Hash	SHA2-512 [8]

NaCl provides primitives shown in Table 1. In addition, NaCl provides APIs which allow one to perform encryption and decryption using NaCl primitives.

NaCl and μNaCl is designed to protect following known vulnerability issues [2, Section3] [3, Section3.1] .

(1) Secret load addresses.

Normal CPU but not embedded one has cache memory and the data in the cache can be accessed faster than those in memory. Attackers use such a time difference for timing attacks. We should avoid secret-key dependent load address, which is called secret load address in [3]. Some microcontrollers such as AVR and ARM Cortex-M0 do not have a CPU cache memory, and they are free from the attack.

(2) Secret branch conditions.

Attackers can perform timing attack by measuring the difference of execution time of each branch. Reasons why such attacks can be performed are the existence of secret-key dependent branch conditions [9] and success/failure of branch prediction [10]. We should avoid secret-key dependent branch conditions, which are called secret branch condition in [3].

In the similar vein, NaCl incorporates the following measures against the attacks.

- Remove the conditional branch that depends on the secret information.
- Make loop counts deterministic.

AVR microcontroller also adopts these measures. AVR and ARM Cortex-M0 microcontroller do not have branch predictor and secure against the attack.

(3) Avoiding unnecessary randomness.

In 2008, it is discovered that OpenSSL generates a predictable random number(CVE-2008-0166) [11]. The cause of the problem is that the code for the OpenSSL random number generation was patched by a wrong code. To avoid the problem NaCl has centralized the random number generation to the OS random number generator [2, Section3 Centralizing randomness] . In addition, NaCl avoids unnecessary use of randomness to reduce the problems arisen from random numbers [2, Section3 Avoiding unnecessary randomness] .

2.2 ARM Cortex-M0 and M0+ Microcontroller

ARM Cortex-M microcontroller series are 32-bit RISC architecture microcontrollers developed by ARM Inc. Cortex-M0 and M0+ have characteristics of low-cost and power-saving and they are expected to be used for the communication of sensor nodes. Hereafter, we simply write Cortex-M0 without describing both of them as long as it does not cause confusion.

Cortex-M0 architecture has thirteen 32-bit general-purpose registers(R0-R12) and three special registers(R13-R15). In particular, R0-R7 are called "low-register", and R8-R12 are called "high-register". Among special register R13-R15, R13 is a stack pointer(SP), R14 is a link register(LR) and R15 is a program counter(PC).

Cortex-M0 uses Thumb instruction set that is 16-bit(half-word) fixed length instruction. The Thumb instruction set has some limitation such that a lot of instruction cannot use high-register. However, we can write highly-efficient and small code using the Thumb instruction set.

In addition, Cortex-M0 has 3-stage(M0+ has 2-stage) pipeline. If it is possible to use these units effectively, it can speed up the execution.

Although Cortex-M0 hardware multiplier is optional, it can calculate 32×32 bit multiplication and output the lower 32-bit result. The execution cycle of the multiplication instruction(MUL) is dependent on multiplier unit implementation. One out of two types, "fast" and "small", of multiplier unit can be implemented at the time of processor manufacturing. The "fast" implementation can perform a multiplication in 1-cycle. Although the "small" implementation requires 32-cycle for multiplication, but its implementation size can be reduced.

3. Implementation

The core part of AVR NaCl is written in about 6,000-line AVR assembly language for optimization and safety improvement. We rewrite this AVR assembly code to ARM Thumb assembly code, and μNaCl is implemented to work on the ARM Cortex-M0. Moreover, we attempt small security fix and optimization.

Main points of our implementation are as follows,

- Adoption of the pseudo-random number generation method whose entropy source is the initial value of SRAM.
- Importing M0NaCl's 256-bit multiplication code with the balance of speed and code size.

Other point of the implementation is the treatment of branch conditions. We remove branch instructions as much as possible for the improvement of speed. The removal also allows us to prevent unknown security breaches caused by branch instructions which have indirect relation with secret key.

3.1 Pseudo-Random Number Generation Method

Two random number generation methods proposed in the paper of AVR NaCl [3, Section3.1 Randomness generation] have the following problems. The first method uses an external random number generator, which increases the cost due to the addition of the external device. The second method uses jitter of RC oscillator [12], for which frequency injection attack [13] has been discovered.

We adopt a pseudo-random number generation method using Salsa20 as encryption scheme and the initial value of the SRAM [14] as a random seed. The process of the pseudo-random number generation is based on that of "arc4random" of OpenBSD, and the encryption scheme ChaCha20 is replaced with Salsa20 and the random seed is replaced with the initial value of the SRAM. The advantage of this method is the use of the existing Salsa20, whose code size has already been reduced and the random seed generation which is low cost and has not yet been attacked.

The procedure of the pseudo-random number generation method is as follows.

1. Get an 12,800-bit initial value of SRAM.
2. Input the initial value to SHA2-512 to create a random seed.
3. Set the random seed to an internal state of Salsa20.
4. Calculate Salsa20 to output a 256-bit pseudo-random number.

3.1.1 Security of Random Number

In μNaCl, random numbers only in 32-byte key generation. We discuss are required entropy required for random numbers. As recommended in NIST SP800-90 [15] entropy of the random number seed should be more than 1.5 times the amount of the shared key. Therefore, the entropy of the random number seed should be more than 384 bits. The initial value of SRAM entropy is 3% per 1-bit [16]. Therefore, 12,800 bits (1,600byte) should be extracted from the SRAM as an initial value. 20,000-bit random numbers generated on MKL25Z128VLK4(Cortex-M0+) by using this method have passed the test of FIPS 140-2.

3.2 Importing M0NaCl's 256-bit Multiplication Code with the Balance of Speed and Code Size

The Curve25519 code of M0NaCl is very fast, but its code size is large. According to [4], the Curve25519 code size is 7,900-bytes, but program flash size of many Cortex-M0 microcontrollers is less than 32-KB. If we use the Curve25519 code of M0NaCl as it is, about 25 % of the program flash are occupied and it becomes difficult to implement other cryptoprimitives and user application codes. Therefore, we import the Curve25519 code of M0NaCl with the following change to save code size.

- Using 256-bit multiplication instead of 256-bit squaring.
- Reimplementation of 256-bit multiplication using 128-bit multiplication.

3.2.1 Reimplementation of 256-bit Multiplication Using 128-bit Multiplication

The 256-bit multiplication code of M0NaCl treats Cortex-M0 multiplication instruction as 16-bit multiplication instruction, and uses three-level Subtract Karatsuba method [4, Section 5.2 Multiplication] . This 256-bit multiplication code uses 128-bit and 64-bit Subtract Karatsuba method multiplications and 32-bit schoolbook multiplication. The cause of large code size is that 256-bit multiplication of M0NaCl does not reuse any long-multiplication code, so these codes are written one by one. Therefore, we reuse 128-bit multiplication for 256-bit multiplication code in order to save code size.

We call "fast" code is the original 256-bit multiplication code, and "small" code is the improved one with saved code size with 128-bit multiplication. The user can select "small" or "fast" according to the purpose. Both benchmark results are given in Sect. 4.1.

4. Benchmark Results

In this section, we show the benchmark results of our implementation described in Sect. 3.

4.1 256-bit Multiplication Benchmark Result

"fast" and "small" 256-bit multiplication codes have the following differences.

We show in Table 2 that "small" code can save its size about half. Table 2 and Table 3 show "small" multiplication code is about 20% slower than "fast" code.

4.2 AVR NaCl versus Cortex-M0/M0+

There are fast version and small version of AVR NaCl. Benchmark results of the fast version of the AVR and our work are shown in Table 5. Note that the benchmark results are obtained in environments shown in Table 4.

The Table 5 shows that this implementation is about 3 times faster than AVR result and reduces half of the code size. Compared with the results of the AVR NaCl, the code size is smaller than even "small" version whose size is 18,328-byte. Theoretically, Cortex-M0 can perform 4 times

Table 2 Benchmark result of "fast" and "small" multiplication

	fast	small
Execution cycle	1350	1633
256-bit multiplication code-size(byte)	2176	436
128-bit multiplication code-size(byte)	-	664
Total of code size(byte)	2176	1100

Table 3 Benchmark result of using "fast" and "small" multiplication

API or Primitives	fast	small	fast / small
crypto_box_curve25519xsalsa20poly1305_keypair	4236162	4960910	85%
crypto_box_curve25519xsalsa20poly1305_beforenm	4215360	4940127	85%
crypto_box_curve25519xsalsa20poly1305[1056-byte]	4390192	5114957	86%
crypto_box_curve25519xsalsa20poly1305_open[1056-byte]	4291964	5016725	86%
crypto_dh_curve25519_keypair	4236154	4960913	85%
crypto_dh_curve25519	4209862	4934628	85%
crypto_scalarmult_curve25519_base	4209866	4934628	85%
crypto_scalarmult_curve25519	4209843	4934603	85%
crypto_sign_ed25519_keypair	4590954	5326281	86%
crypto_sign_ed25519	5119331	5854948	87%
crypto_sign_ed25519_open	8764402	9923302	88%

Table 4 Benchmark environment

	AVR NaCl	Our work	
		Cortex-M0+	Cortex-M0
Board	Arduino MEGA2560	FRDM-KL25Z	mbed LPC11U24
Microcontroller	ATMEGA2560	MKL25Z128VLK4	LPC11U24
Source code	avrnacl-20140813, fast	-	-
Compiler	gcc 4.8.1, -O3	gcc 4.8.3, -O2	gcc 4.8.3, -O2

Table 5 μNaCl benchmark result

API	Message bytes	Execute Cycle			Gain AVR / M0+	Stack bytes
		AVR	ARM			
			Cortex-M0+	Cortex-M0		
crypto_auth_hmacsha512256	1024	6367104	2730411	3818420	233%	940
crypto_auth_hmacsha512256_verify	1024	6367512	2730769	3818951	233%	940
crypto_box_curve25519xsalsa20poly1305_keypair		23239860	4236162	5193420	549%	516
crypto_box_curve25519xsalsa20poly1305_beforenm		22853464	4215360	5165998	542%	532
crypto_box_curve25519xsalsa20poly1305	1056	23342860	4390192	5390007	532%	596
crypto_box_curve25519xsalsa20poly1305_open	1056	23061840	4291964	5263031	537%	596
crypto_box_curve25519xsalsa20poly1305_afternm	1056	489830	174859	224058	280%	596
crypto_secretbox_xsalsa20poly1305	1056	489812	174845	221060	280%	476
crypto_secretbox_xsalsa20poly1305_open	1056	208536	76652	96734	272%	476
crypto_hash_sha512	1024	4773346	2046102	2835995	233%	884
crypto_dh_curve25519_keypair		23239836	4236154	5192245	549%	516
crypto_dh_curve25519		22836606	4209862	5157930	542%	516
crypto_stream_salsa20	1024	266092	87292	107253	305%	364
crypto_stream_salsa20_xor	1024	281300	98564	123578	285%	364
crypto_stream_xsalsa20	1024	282728	92809	113988	305%	412
crypto_stream_xsalsa20_xor	1024	298210	104095	130326	286%	412
Primitives						
crypto_core_hsalsa20		17019	5527	6863	308%	268
crypto_core_salsa20		16930	5444	6768	311%	204
crypto_hashblocks_sha512	1024	4239316	1817297	2517098	233%	532
crypto_onetimeauth_poly1305	1024	173767	64937	81257	268%	292
crypto_onetimeauth_poly1305_verify	1024	174005	65141	81549	267%	292
crypto_scalarmult_curve25519_base		22836588	4209866	5164352	542%	500
crypto_scalarmult_curve25519		22836588	4209843	5164176	542%	492
crypto_sign_ed25519_keypair		21913629	4590954	5678887	477%	1580
crypto_sign_ed25519	1024	22691528	5119331	6435293	443%	1676
crypto_sign_ed25519_open	1088	37562656	8764402	11100947	429%	1676
crypto_verify_16		383	236	338	162%	12
crypto_verify_32		553	396	578	140%	12

NaCl implementation	Code size(bytes)		Difference
	26924	15704	11220

faster with only twice longer instructions than AVR microcontroller. The results shown above mean that we have succeeded in achieving about 70% of the efficiency in our implementation. In Table 5 the scores of programs related to Curve25519 are high. This is because the code of M0NaCl has been used.

Table 5 shows that stack size used in this library is less than 2000-bytes. We expect the effect of footprint for utilizing the software library is enough small.

5. Concluding Remarks

In this paper, we implemented μNaCl in Cortex-M0 microcontroller for all primitives of M0NaCl, only Curve25519 of which has been implemented so far [4], and showed that the implementation is 3 times faster than AVR NaCl. So far, we have not yet succeed in optimizing all of the codes. We can expect that μNaCl with higher-speed can be obtained after the optimization.

Entropy contained in the SRAM has been estimated from the results for STM32 microcontroller [16]. Since there are many ARM microcontrollers e.g. NXP, FreeScale, Atmel, we should examine the amount of entropy contained in the SRAM of those microcontrollers.

If we can port this library into mbed environment, then the security of many products using mbed can be improved and many people can get benefit of the library. In the mean time, program flash size is less than 32-KB in many Cortex-M0 microcontrollers supported by mbed. The program flash size is not large enough for installing our implementation with basic libraries, e.g. libgcc and libstdc++. The balance of speed and size is quite important in the implementation. A library having an appropriate balance be widely used in mbed and other Cortex-M0 microcontrollers.

Concerning preemptive multitasking, we expect that preemptive multitasking is possible if the scheduler saves registers and restore it correctly. Basically, the NaCl uses only stack. Few global memory may be used but its use is limited to store only read-only values e.g. SHA2-512's constant. Therefore, even if another process executes NaCl's encryption operation during the interruption of main process, the process does not influence to the main process. Since we have not yet fully checked the behaviors of preemptive multitasking in our environment, these issues are remained as the future work.

Acknowledgments

A part of this research is supported by JSPS A3 Foresight Program.

References

[1] mbed webpage. https://www.mbed.com/en/
[2] D.J. Bernstein, T. Lange, and P. Schwabe, "The security impact of a new cryptographic library," Progress in Cryptology – LATINCRYPT 2012, vol.7533, pp.159–176, Springer, 2012.
[3] M. Hutter and P. Schwabe, "NaCl on 8-Bit AVR Microcontrollers," Progress in Cryptology – AFRICACRYPT 2013, vol.7918, pp.156–172, Springer, 2013.
[4] M. Düll, B. Haase, G. Hinterwälder, M. Hutter, C. Paar, A.H. Sánchez, and P. Schwabe, "High-speed curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers," Designs, Codes and Cryptography, vol.77, no.2-3, pp.493–514, 2015.
[5] D.J. Bernstein, "Curve25519: new Diffie-Hellman speed records," Public Key Cryptography-PKC 2006, vol.3958, pp.207–228, Springer, 2006.
[6] D.J. Bernstein, "The Salsa20 family of stream ciphers," in New Stream Cipher Designs, pp.84–97, Springer, 2008.
[7] D.J. Bernstein, "The Poly1305-AES message authentication code," Fast Software Encryption, vol.3557, pp.32–49, Springer, 2005.
[8] FIPS PUB, Secure Hash Standard (SHS), 2012.
[9] B.B. Brumley and N. Tuveri, "Remote timing attacks are still practical," Computer Security – ESORICS 2011, vol.6879, pp.355–371, Springer, 2011.
[10] O. Acıiçmez, Ç.K. Koç, and Jean-Pierre Seifert, "Predicting secret keys via branch prediction," Topics in Cryptology – CT-RSA 2007, vol.6879, pp.225–242, Springer, 2006.
[11] Inc. Software in the Public Interest, Debian security advisory, DSA-1571-1 openssl—predictable random number generator, 2008.
[12] J. Hlavác, R. Lórencz, and M. Hadácek, "True random number generation on an Atmel AVR microcontroller," Computer Engineering and Technology (ICCET), 2010 2nd International Conference on, vol.2, pp.V2-493–V2-495, IEEE, 2010.
[13] S. Buchovecká and J. Hlaváč, "Frequency injection attack on a random number generator," IEEE 16th International Symposium on DDECS'13, pp.128–130, 2013.
[14] A. Van Herrewege and I. Verbauwhede, "Software Only, Extremely Compact, Keccak-based Secure PRNG on ARM Cortex-M," Proceedings of the 51st Annual Design Automation Conference, DAC '14, pp.1–6, New York, NY, USA, ACM, 2014.
[15] J. Barker and E. Kelsey, NIST Special Publication 800-90A: Recommendation for Random Number Generation Using Deterministic Random Bit Generators, Jan. 2012.
[16] A. Van Herrewege, V. van der Leest, A. Schaller, S. Katzenbeisser, and I. Verbauwhede, "Secure PRNG Seeding on Commercial Off-the-shelf Microcontrollers," Proceedings of the 3rd International Workshop on Trustworthy Embedded Devices, TrustED '13, pp.55–64, New York, NY, USA, ACM, 2013.

Toshifumi Nishinaga graduated School of Electrical and Computer Engineering, College of Science and Engineering, Kanazawa University in 2015. He is now a master course student of Division of Electrical Engineering and Computer Science, Graduate School of Natural Science and Technology, Kanazawa University. He has been engaged in research and development of embedded systems.

Masahiro Mambo received a B. Eng. degree from Kanazawa University, Japan, in 1988 and M.S. Eng. and Dr. Eng. degrees in electronic engineering from Tokyo Institute of Technology, Japan in 1990 and 1993, respectively. After working at Japan Advanced Institute of Science and Technology, JAIST, at Tohoku University and at University of Tsukuba as assistant, associate and associate professor, respectively, he joined Kanazawa University in 2011. He is currently a professor of Faculty of Electrical and Computer Engineering, Institute of Science and Engineering. His research interests include information security, software protection and privacy protection.