TRS-80 With A Grain Of Salt

Jean-Marie Chauvet

MassiveRand
jmc@massiverand.com
http://www.massiverand.com
62, ave. Pierre Grenier, 92100 Boulogne-Billancourt, France

Abstract. This paper presents early results of a (very) experimental implementation of the elliptic curve and stream cipher calculations of the Networking and Cryptography library (NaCl), on the TRS-80 Model I. Needless to say, the demonstration that such a library, which has been optimized for many modern platforms including leading edge desktops, servers and, recently, modern microcontrollers, is even feasible on such early home microcomputers is, at best, to be considered a recreation rather than as a practical application of technology. In the process, however, lessons were learned in implementing trade-offs for basic cryptographic primitives and, more importantly maybe, in experimenting with some transformative aspects of retrocomputing.

Keywords: Cryptography, Random Bit Generator, ECC, Stream Cipher, ChaCha, Curve25519, Z80, TRS-80, Retrocomputing, Crazy Ideas, Remix

1 Introduction

This paper briefly reviews an experimental implementation of the key generation and stream cipher functions of the Networking and Cryptography Library (NaCl) [6] on the TRS-80 Model 1, first released in August 1977. On such a limited resources platform, by today's standards, the aim is obviously not to achieve high speed, as in current ports of the library to various modern CPUs, but rather to achieve some decent execution time within the diminutive footprint.

More specifically, we report on an implementation of the Chacha stream cipher [4], and the Curve25519 elliptic-curve Diffie-Hellman key-exchange protocol [3]. It provides the basis for a complete port including the Poly1305 authenticator [2] and the Ed25519 elliptic-curve signature scheme [5]. In confronting this retrocomputing challenge, we drew solace from the NaCl designers claim that:

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

and inspiration from the recent full ports of NaCl to 8-bit AVR microcontrollers [9].

2 The Z80-based TRS-80 Model I

It was with minimal expectations that, in August 1977, Tandy Corporation teamed up with Radio Shack to release the TRS-80, one of the first personal computers available to consumer markets. And as it turned out, the TRS-80 surpassed even the most cautious sales estimates by tenfold within its first month on the market despite a hefty $600 price point; the burgeoning prospects of a new era in personal electronics and computing could no longer be denied. While the original machine shipped with BASIC Level I, based on Wang's free TinyBASIC, and 4KB of memory, a 12" monitor and a Radio Shack tape recorder as datacassette storage, later shipments in 1978 came with Level II BASIC, licenced from Microsoft, and 16KB of memory: this is the target platform of the present implementation. (The ability to expand memory up to 48KB, to access floppy drives, a second cassette port, a Centronics parallel printer and an optional RS232 port required the proprietary Expansion Interface, a bulky box that fit under the monitor.) The TRS-80 uses a Zilog Z80 CPU clocked at 1.77MHz.

All data in the TRS-80 and most in the Z80 is handled in (8-bit) bytes. There are 14 general purpose registers in the CPU, designated A, B, C, D, E, H and L and the so-called primed counterparts, A' to L'. At any given time one set, primed or non primed, is active. The A register is used as a default operand in many of the instructions and is often referred to as the accumulator. Registers are paired, as BC, DE, HL and AF, in the few 16-bit arithmetic and pointer operations in the instruction set. Special-purpose registers are the 16-bit program counter, PC, the 16-bit stack pointer, SP, and two 16-bit index registers, IX and IY. The flag registers are F, F' respectively, 8 bits signalling sign, zero, half-carry, parity/overflow and carry after arithmetic and logic operations [12]. The Z80 instruction set, a superset of the older 8080 and 8008 instruction sets, affords more than 500 combinations altogether covering data movement, arithmetic, logical and compare, decision-making and jumps, stack operations, bit shifting and I/O operations [10]. Note that in contrast to modern microcontrollers such as e.g. the ATmega and ATxmega cores family from Atmel, there is no multiplication instruction–not necessarily an auspicious starting point for public-key cryptography.

We use the representation of large integers and elements of finite fields as byte arrays using radix 2^8 typical on such 8-bit architectures.

3 The NaCL library, re-contextualized

The *Networking and Cryptographic Libray*, NaCL pronounced "salt", was one deliverable of the European Commission funded project CACE (2008-2010). Ulterior developments were continued within ECRYPT II, the European Network of Excellence in Cryptology. Its well-known features are its high-level API for public-key authenticated encryption and cryptographic signatures; its high-level of security and protection against timing attacks; its record speed on modern architectures; and its availability as free of copyright restrictions. The library was precisely designed to these characteristic features in a situated context as reflected upon by its authors [6], namely repeated breaches of confidentiality and integrity on the Internet, in spite of widely accepted standards such as AES-128 and RSA-2048 and publicly available implementations such as OpenSSL.

Our two-pronged approach aims primarily at porting as much as the NaCl functions to the TRS-80, but also at reassessing these contextualized features in a different context, the early home-computing subculture–now arguably extinct or viewed as akin to historical preservation. As much as the architecture of the Internet provides today the context for the need and design of the NaCl libray, the TRS-80 platform and, more broadly speaking, the practices of the then community of its users should provide the background context of this retrocomputing variant of the modern library.

The implementation offers scalar multiplication on the elliptic curve Curve25519 [3] and a stream cipher. Instead of the Salsa20 stream cipher in the original design [6] we chose to implement a variant, ChaCha [4], which has also recently been successfully implemented on FPGA [1]. Obviously porting cryptographic functions to a memory-restricted architecture such as the original Model 1 with 16KB, requires high reuse of code across all primitive operations in order to minimize its size. Generic 32-byte addition, subtraction and, crucially, multiplications are coded in assembly language and shared across all modular arithmetic operations, some of which are written in C. On the other hand, issues of secret branch conditions and secret load addresses, for instance, often sources of timing attacks, are somewhat subdued in this implementation as there is no cache optimization nor branch prediction technique. Still, we kept to the strategy exemplified by other NaCl implementations on modern architectures avoiding leaking secret data through branch conditions.

Entropy pool and randomness generation presented quite another interesting contextual issue. NaCl uses the OS-provided random number generator (RNG), reading random bytes from `/dev/urandom`. The BASIC Level II interpreter in ROM [11] features a pseudo-RNG through two calls: `RANDOM` to seed the PRNG, and `RND(N)` which returns a single-precision floating point value between 0 and 1 if N is 0; a random integer between 1 and N, included, when $N > 0$. After some disassembling of the ROM and further interacting with the TRS-80 discussion groups, it surfaced that this PRNG is a Linear Congruential Generator based on the following iteration: $x_n = 4253261x_{n-1} + 372837 \quad mod \quad 2^{24}$. The seed (24-bit long) is stored at a specific lower RAM address where `RANDOM` overwrites its middle byte with the Z80 internal R register. There is no seed initialization at power-up or reset, so that failure to call `RANDOM` causes the same sequence of random numbers to be produced in successive calls to `RND`. Interestingly enough, the same LCG was apparently already present in the older Level I, which was based, however, on a completely different BASIC, namely TinyBASIC. A 2^{24} modulus LCG is also present in the later QuickBasic, GW Basic (with constants 214013 and 2531011), and even in Visual Basic 6 (with constants 1140671485 and 12820163) all released by Microsoft in the following decade. Hardly cryptographically secure entropy pools!

Of course, one can use the implementation, like NaCl for that matter, in a way that doesn't require local randomness simply by generating key pairs on an external device and transferring them on the TRS-80. Which of course leads us right into the central role of cassettes as both data storage and data communication (audio) channel–a topic we return to in the last section. In light of the famously erratic results of cassette operations for programs (`CLOAD`, `CSAVE`) and for data (`PRINT#`)–a hallmark "feature" of the Model 1–an alternate exploration is to fiddle with the low-level I/O instructions of the Z80 (`IN a,(n)`, `OUT (n),a`) to sample some noisy bytes from a vintage *audio* cassette [8]. (The author's vintage cassette edition of *Rumours* by Fleetwood Mac worked well, but David Bowie and Foreigner also yielded interesting results.)

The question of the BASIC API is important in maintaining at least a thin varnish of authenticity to the effort, as this language became the prominent instrument in the programming usage the budding home-computing community engaged in [7] at the time. We kept it to a rough minimum, however, as constrained memory didn't really allowed much sophistication. The cryptographic functions are machine language routines first read from (again) cassette through the `SYSTEM` command. The BASIC Level II commands `POKE I,B` and `PEEK(I)` respectively sets the value of memory location I to B and returns the value at memory location I. The messages to be encrypted, as well as the keys, if generated externally, are thus "poked" individually and byte-wise at determined locations in memory. The machine language routines themselves are called using the `USR(N)` idiom in BASIC Level II, once the entry point of the selected routine has been "poked" in reserved addresses 16526 and 16527, LSB first–such an idiosyncratic procedure that it remains a highlight of the TRS-80 folklore that no user could ignore.

The tool chain is the instrument of port and recontextualization: we used the Open Source SDCC suite of tools (on SourceForge: sdcc.sourceforge.net), which provides a C compiler and a Z80 assembler/linker, and a Z80 emulator. We also used ZEMU, a Z80 emulator by Joe Moore; z80dasm, a disassembler; and several TRS-80 emulators (sdltrs, available at sdltrs.sourceforge.net, and TRS32 from www.trs-80emulators.com). Critical to the success of the implementation is of course Knut Roll-Lund's pair of programs *playcas* and *wav2cas* which translate to and from the audio format as recorded on tape to digital binaries, effectively allowing replacement of the cassette player with a PC hooked through the audio/microphone jacks. We created a few additional Python scripts to translate from the standard IHX format to the custom cassette binary format as specified in the TRS-80 original manuals [11]. And, of course, tests were ran on the author's loyal 16K machine from 1978. Note that most of these tools and utilities are often the passionate work of hobbyists producing–perhaps disconcertingly–retrocomputing "assemblages" involving an admixture of technologies from different historical periods.

4 Implementing ChaCha

The ChaCha state matrix is simply represented as an array of 64 bytes. The cipher consists of 8 rounds that alter this state through logical and arithmetic transformations. In ChaCha these operations (plus, xor and bit rotations) apply to 32-bit integers. Here we coded the addition and the bit shuffling in assembly language in order to operate on 4 bytes at a time. Constants, nonces and message slices are entered byte-wise either in the core machine language for constants or by "poking" from the BASIC minimal API. Once initialized, the cipher calls 4 times a sequence of eight quarter-rounds on four 4-byte state elements.

Having a minimal number of registers available, the bit operations implementation resorts to storing and loading from global memory, heavily using the IX and HL registers. Overall one quarter-round function call requires 8257 T-states. (The Z80 machine cycles are sequenced by an internal state machine which builds each machine cycle out of 3, 4, 5 or 6 T-states depending on context.) The entire round of calculation needs 266,688 T-states. The code size of ChaCha is 1,465 bytes and uses 74 bytes of data memory. Execution time clocked on 16K TRS-80 is about 0.3 s.

5 Implementing Scalar Multiplication on Curve25519

The main computational effort in the Curve25519 elliptic-curve Diffie-Hellman primitive [3] is the scalar multiplication using the x-coordinate based differential Montgomery addition. All arithmetic is done over the field $F_{2^{255}-19}$. The scalar multiplication consists of 255 so-called ladder steps, 255 conditional swaps, each one based on a single bit of the scalar, and one final inversion to recover the final x coordinate. Each ladder step, in turn, is a sequence of 13 multiplications including squarings, 4 constant multiplications (by 4, and by 486,662), 3 additions and 3 subtractions on arrays of 32 bytes.

The most speed-critical operations are multiplications and squarings which we didn't specialize to reduce code size. The multiplication of 2 byte-sized integers with accumulation into a 4-byte integer (MULADD) is implemented in assembly. It is used as the core operation in a straightforward implementation of a 32-byte by 32-byte Comba multiplication. On top of the multiplication we need several additions, implemented in assembly: adding a 1-byte and a 2-byte integer to a 32-byte integer with carry propagation (ADD013233 and ADD023233); adding two 32-byte integer with carry (ADD323233) and one 32-byte to a 33-byte integer (ADD323333); and subtration of 32-byte integer from a larger 32-byte integer (SUB323232). The entire Comba multiplication uses 558,811 T-states and 1,024 calls to MULADD.

Throughout the scalar multiplication we reduce modulo $2^{256} - 38$. After a multiplication, reduction is obtained by splitting for the 32-byte MSB part of the result, multiplying it by 38 and adding to the 32-byte LSB (MUL38x3233 followed by ADD323333). As carry may be propagated we reiterate multiplying the MSB by 38 and adding to the LSB, choosing the appropriate addition implementation from the set mentioned in the last paragraph (i.e. ADD023233 on each iteration). After and addition or subtraction, only the last steps are required since the carry is 1-byte long (i.e. the ADD023233 iterations). The final inversion in $F_{2^{255}-19}$ is performed as an exponentiation to $2^{255} - 21$ using the addition chain of of 254 squarings and 11 multiplications found in the reference implementation [3]. Finally we reduce to $P = 2^{255} - 19$ by subtracting P once if the value is greater then P (as it is, by definition, less than $2P$).

The scalar multiplication performs 3,562 calls to the modular multiplications which resolve to 3,647,488 calls to the MULADD core; the reciprocal performs 266 and 272,384 calls respectively. A complete modular multiplication use an average of 733,244 T-states. Needless to say, for a machine clocked at 1,77 MHz this translates into prohibitive execution times. We measured an average of 44 minutes (!) for scalar multiplication and a comparatively swift 4 minutes for modular inversion on an original 16K TRS-80, bringing key generation to a respectable total time of 48 minutes. The total code size is 7,880 bytes and uses an additional 733 bytes of globals.

6 Conclusions

The implementation presented here exemplifies some aspects of the world of retrocomputing, a set of diverse practices involving contemporary engagement with old computer systems. As testified by performance measurements, this version of a NaCl subset definitely cannot aim at the highest speed, even on 8-bit architectures. Performance of modern 8-bit platforms such as current microcontrollers are already many steps higher than afforded by the Z80-based TRS-80. Moreover it does not pretend to be optimal. There is indeed ample room for optimization in this first-pass implementation (e.g. looking at special code for squarings, better operand caching in spite of the limited number of registers...).

However, in contrast to retrocomputing recognized as being simply preservation – although maintaining an original TRS-80 with monitor and tape deck in working condition to this date requires constant attention – the current work illustrates a hobbyist approach to remix and resituating. For instance, while both the RSA publication and the TRS-80 general release date back from 1977, the use of elliptic curves in cryptography is posterior (1985), and the choice of elliptic curve with proper formulas for fast Diffie-Hellman resulted from research work posterior to 2000. So there is a transformative aspect in this assemblage of elements from different historical periods, a novel recombination, remixing theory published in the mid-eighties with applied considerations from the early 2000s jammed into a vintage late-seventies machine, which may simply seem like an authentic artifact from the period to a naive observer. Another hybridization worth noting is the use of an ad-hoc application on contemporary PCs to "play" programs stored in the original cassette data format. This is a short step to preserving TRS-80 programs and data, old or new, on-line, where this TRS-80/modern PC collage effectively allows CLOAD from the cloud.

Several features of the selected cryptographic library are important to today's applications in the current data and network security context. It is important to resituate or map these features into the characteristic identity feature set of the original machine. Hence the BASIC-driven interface and the idea of exchanging public keys by exchanging cassette tapes, which were typical practices then in programming for the TRS-80 and in distributing programs.

Acknowledgements

The author wishes to thank Neil Morrison, George Phillips, and Al Petrofsky of the TRS-80 Yahoo! group for their wise advice and knowledgeable help in the historical investigation of random number generation in BASIC dialects.

References

1. Nuray At, Jean-Luc Beuchat, Eiji Okamoto, Ismail San, and Teppei Yamazaki. Compact Hardware Implementations of ChaCha, BLAKE, Threefish, and Skein on FPGA. Cryptology ePrint Archive, Report 2013/113, 2013. `http://eprint.iacr.org/`.
2. Daniel J. Bernstein. The Poly1305-AES Message-Authentication Code. In *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, 2005.
3. Daniel J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer, 2006.
4. Daniel J. Bernstein. ChaCha, a variant of Salsa20. In ECRYPT, editor, *SACS 2008 The State of the Art of Stream Ciphers*, Information Society Technologies, page 273. 2008.
5. 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, *CHES*, volume 6917 of *Lecture Notes in Computer Science*, pages 124–142. Springer, 2011.
6. Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. Cryptology ePrint Archive, Report 2011/646, 2011. `http://eprint.iacr.org/`.
7. David Brin. Why Johnny can't code. *Salon.com*, 2006.
8. Pierre Giraud and Alain Pinaud. *La pratique du TRS-80*, volume 2. Editions du P.S.I., 1980.
9. Michael Hutter and Peter Schwabe. NaCl on 8-Bit AVR Microcontrollers. In Amr Youssef, Abderrahmane Nitaj, and AboulElla Hassanien, editors, *Progress in Cryptology AFRICACRYPT 2013*, volume 7918 of *Lecture Notes in Computer Science*, pages 156–172. Springer Berlin Heidelberg, 2013.
10. Lance A. Leventhal. *Z80 Assembly Language Programming.* Osborne and Associates, 1979.
11. Edwin R. Paay. *Level II ROM Reference Manual.* MICRO-80 Products, 1980.
12. Jr. William Barden. *TRS-80 Assembly-Language Programming.* Radio Shack, 1979.