

HASHLIB

Industry-Standard
Cryptography
on the
TI-84+ CE

Analysis & Extended Information

by Anthony Cagliano

Contents

1	Secure PRNG Construction & Analysis	2
1.1	Construction Overview	2
1.2	Correlation in the Bit Stream	2
1.3	Proof of Cryptographic Security	3
2	Side-Channel Analysis	4
2.1	Timing Analysis Protection	5
2.2	Buffer Leak Protection	5
2.3	Memory-Mapping Protection	6
3	Algorithmic Security	7
3.1	CPA Resistance	7
3.2	CCA Resistance	7
4	Conclusion	7
5	References	8

Foreword

Despite being developed for an unconventional platform, HASHLIB implements some of the most secure encryption in use today with an API capable of generating secure keys and salts, as well as constructing secure channels of communication between the device and a host computer or server. The library facilitates ciphertext obfuscation and integrity between two endpoints, and also allows for authentication and identity verification between said endpoints. This document is a technical descriptor, providing details relevant to the construction of algorithms within HASHLIB that were built from the ground up, as well as details hardening measures taken to secure the library as much as possible. This document is provided such that platform-specific components of the library are able to be analyzed and peer-reviewed. For basic help with using the library, including overviews of defines, macros, and functions, consult the separate *HASHLIB Quick Reference* document, which is more geared towards general library documentation.

I would like to offer a special thanks to the members of the Cemotech programming whose support and assistance has made the continuing development of this library possible. Not only have a few members been active contributors to this project's code-base, but many others have contributed to discussions on SAX and Discord about entropy, math and proofs, and methods of hardening the library against attack.

1 Secure PRNG Construction & Analysis

1.1 Construction Overview

The **SPRNG** is the most platform-dependent part of this code, and consequentially the part of it the most work was done on. The first question to answer in the quest to devise one was does the TI-84+ CE produce randomness, and if so, how does this differ by hardware revision. The calculator is unlike a computer in that it lacks many of the dynamic sources of entropy that go into generating randomness for those platforms. To this aim, the **Cemetech** user **Zeroko**¹ was of much assistance. At their direction, I discovered that the address range \$D65800 through \$D659FF contains hardware “bus noise”, and that certain parts of this address space are more heavily biased than others and that these sections differ by revision. This helped lay the foundation for an algorithm that polls each bit of every byte in this address space looking for the bit with the least bias (out of N bit reads, the bit was set closest to N/2 times). The most optimal byte is set internally, and is unable to be modified by the user. This allows the **SPRNG** to utilize the same code regardless of revision, without having to use a revision-based lookup table that might not work on all software (or hardware) revisions. Should a bit of sufficient entropy not be found (the maximum allowable deviation in the set bit count is $N/2 \pm 25\%$ of N, where N is the total number of bit reads), the function will return **NULL**, and calls to the other **SPRNG** functions will return either 0 or **NULL**.

The **SPRNG** reserves an entropy pool 119 bytes large in the device’s accelerated RAM (to make the generator faster). To generate a random number, each byte in the entropy pool is updated by reading from the selected entropic byte. The entire pool is then passed through the SHA-256 cryptographic hash, generating a 32-byte digest. That hash is broken into 8-byte blocks, each byte of which is xored together to produce a single byte. The result is a 4-byte compression of the 119 byte entropy pool.

1.2 Correlation in the Bit Stream

Due to the nature of the CE hardware, there is measurable correlation in the values of floating bits within unmapped memory. This correlation varies depending on the sample size, but would measurably reduce the entropy of the system. While specific numbers on the degree of correlation are, as of yet, unavailable, we do know that there are higher amounts of correlation in the initial reads which decreases after a certain number of reads. To combat this, the entropy gathering function was rewritten with direction from Zeroko. The operating premise of the modification is this: By increasing the number of reads that go into creating the entropy pool, the footprint of the correlation relative to the available entropy will decrease. The modification to the **SPRNG** has each byte of the entropy pool be a composite of seven (7) distinct reads from the entropic byte xored together.

¹<https://www.cemetech.net/forum/profile.php?mode=viewprofile&u=29638>

1.3 Proof of Cryptographic Security

In order to be secure (at least on paper), a PRNG has to pass 2 distinct additional unpredictability assessments in addition to the requirement that it also pass general statistical randomness tests. The two additional tests are the **next-bit test** and the **state compromise test**.

Next-Bit Test

Given the prior output of the PRNG (bits 0 -> i), the next bit (i+1) of the output cannot be predicted by a polynomial-time statistical test with a probability non-negligibly greater than 50%.

State Compromise Test

An adversary gaining knowledge of the initial state of the PRNG does not gain any information about its output.

To solve the next-bit test, I'll make a few assumptions. Firstly, a secure PRNG must pass all statistical tests. A cryptographic hash, like SHA-256, produces output that satisfies this constraint, regardless of the entropy of its input. Second, each output of the PRNG is a 32-bit or 4 byte unsigned integer, meaning we need at least 32 bits of entropy to generate such an integer. Do we have that? Observe the computation below that plugs the worst-case bias in the source byte into a Shannon entropy construction to calculate the available entropy across the 119 bytes of the entropy pool.

Proof. Entropy for Worst-Case Byte Selection

The byte selected for use by the CSPRNG will have at least one bit with a maximum bias of 75% ($\pm 25\%$ deviation from 50/50), with the rest of the byte likely having a much higher bias. To this end, we will assume 100% bias, or 0 entropy, for them. The entropy of the 75/25 bit can be represented as the set of probabilities .75, .25, which can be applied to the equation for Shannon entropy:

$$H(bit) = \sum_{i=1}^n P(i) \log_2 \frac{1}{P(i)}$$

Where n is the length of the set of probabilities and $P(i)$ is the probability of the i -th element in the set

$$H(bit)_{min} = 0.750 \log_2 \frac{1}{0.750} + 0.250 \log_2 \frac{1}{0.250} = 0.811$$

Assuming that this is the entropy of the bit selected, and also assuming that the rest of the byte in question has minimal entropy (whether this is true or not depends on what byte we are selecting. Sometimes there is some entropy in

other bits of the byte, other times the bit values are predictable), the entropy for the entire byte (worst-case) can be calculated like so:

$$H(\text{byte}) = 0.811 + (7 * 0) = 0.811$$

The entropy pool consists of 119 bytes, so to compute the available entropy we multiply the entropy per byte by the size of the pool.

$$H(\text{pool})_{\min}(119) = 0.811 \times 119.0 = 96.51$$

This means we have 96.51 bits of entropy per 32-bit number at minimum, more than double what is needed. Because the available entropy exceeds the bit-width of the number we are generating, I assert that the probability of $r \leftarrow \{0,1\}$ is within a negligible deviation, ϵ , from $\frac{1}{2}$ for all bits in the 32-bit number. This would also mean that the advantage of an adversary attempting to differentiate the output of this SPRNG from a uniform random distribution would also be negligible.

Having proved sufficient entropy in the input, I assert that the output of the generator satisfies all algorithmic constraints for cryptographic security and will now move onto the second test, state compromise. Each output of the generator is based on a freshly-constructed entropy pool, meaning that there is no computational relationship between subsequent invocations of the PRNG. This means that there is no "initial state" of the generator (apart from what byte we are reading entropy from, and leaking that reveals nothing of value), and even the leaking of a prior state of the generator will not compromise the next output. Thus, I assert that this generator also passes the state compromise test, and is thus safe for use as a cryptographic PRNG.

2 Side-Channel Analysis

HASHLIB has an unknown level of resistance to side-channel analysis. The TI-84+ CE is not a device constructed with hardware security in mind, despite TI repeatedly trying to improve security by taking actions that have nothing to do with security. One such example was disabling native assembly code execution to fix a bug that had already been resolved just to appease a bunch of bureaucrats who were well-informed enough to know the bug existed but not well-informed enough to know that it had already been patched. This led to greater efforts at breaching, and exposing, ways to execute code anyway—and not everyone who now has access to that exploit would have the same benevolent intent that myself and the rest of the TI programming community does. Pro-tip: allowing non-field experts to control policy is a really good way to get your policy wrong. You don't see archaeologists controlling NIST cybersecurity standards. A bit of advice for TI if any of you wind up seeing this— You don't nuke an entire feature set to deal with one specific, unrelated, vulnerability. It's more to your advantage to re-enable assembly execution and rethink exam security specifically.

With that rant over, let's circle back to the subject matter at hand—HASHLIB's resistance to side-channel analysis. There are very limited ways in which a device designed to be plugged into a computer and memory-mapped can be protected against straight-up being plugged into a computer and memory-mapped. I took some actions that could reasonably be taken without slowing down the library's functions to harden it against these types of analysis.

2.1 Timing Analysis Protection

One of the first considerations in this library was resistance to timing analysis. While I have not as of the time of this writing evaluated the ported functions, all functions written from scratch for this platform had this form of hardness in mind. The buffer comparison function and the modular exponentiation functions are two examples of this. Over time more of the library will be reviewed and, where necessary, the same considerations will be extended to those functions as well, where necessary.

2.2 Buffer Leak Protection

Another consideration taken within HASHLIB was to avoid leaving residual computational data in the stack frame after a function that performs data transformations completes. For this reason, some code was written that purges the stack frame and we call that code in most of the user-facing encryptor functions before returning control to the caller. For reference, the code used to accomplish this is:

Listing 1: stack purging code by Zeroko

```

1 ?stackBot    := 0D1987Eh
2 stack_clear:
3     ; backup hl, a, and e
4     ld (.smc_a), a
5     ld (.smc_hl), hl
6     ld a, e
7     ld (.smc_e), a
8
9     ; set from stackBot + 4 to ix - 1 to 0
10    ; ix points to the current top of stack frame
11    lea de, ix - 2
12    ld hl, -(stackBot + 3)
13    add hl, de
14    push hl
15    pop bc
16    lea hl, ix - 1
17    ld (hl), 0
18    lddr
19
20    ; restore a, hl, e
21    ld e, 0
22    .smc_e:=$-1
23    ld a, 0
24    .smc_a:=$-1

```

```

25     ld hl, 0
26     .smc_hl := $-3
27     ld sp, ix
28     pop ix
29     ret

```

2.3 Memory-Mapping Protection

After resolving buffer leaks, our attention turned to thwarting attempts to map the device’s memory while HASHLIB is running to glean information about the encryption or decryption. While there is sufficient difficulty in achieving this in a device designed to be used in that manner in other contexts, I believe the solution arrived at is sufficient.

It is the system interrupt that makes it possible for the operating system to handle activity on the USB port, and disabling said interrupts would severely hamper attempts to read HASHLIB’s state by preventing the system from responding to USB transfers. For this reason, some code was written that disables interrupts in all functions where data is being encrypted or decrypted, saving their state to SMC, and restoring that state afterwards. While there are many variations of this code that operate in slightly different ways depending on when they are called, here is the basic version of it, for reference:

Listing 2: disable interrupts code by beekadamtheinventor

```

1 ; helper macro for saving the interrupt state, then disabling interrupts
2 macro save_interrupts?
3     ld a,i
4     push af
5     pop bc
6     ld ().__interrupt_state, bc
7     di
8 end macro
9
10 ; helper macro for restoring the interrupt state
11 macro restore_interrupts? parent
12     ld bc, 0
13 parent.__interrupt_state = $-3
14     push bc
15     pop af
16     ret po
17     ei
18 end macro

```

3 Algorithmic Security

3.1 CPA Resistance

All of the encryption functions in HASHLIB facilitate CPA security by allowing for the inclusion of a nonce (random string of bytes) in the encryption. This nonce ensures that if you grant an adversary the ability to request encryptions of chosen plaintexts (an attack called *chosen plaintext attack*), the ciphertext will not be mappable back to a plaintext due to the presence of randomness in the input. In AES, the randomness is included through the use of an *initialization vector*, a buffer of length equal to the AES block size that is filled with random data. In RSA, the randomness is added via the optimal asymmetric encryption padding scheme (OAEP) which xors the entire message with a salt using a Feistel network.

3.2 CCA Resistance

HASHLIB provides an API that allows you to construct an authenticated encryption protocol that guards against an adversary able to request decryptions of a chosen ciphertext (an attack named *chosen ciphertext attack*). The availability of cryptographic hash as well as a hash-based message authentication code (HMAC) provides this functionality. You can use a hash or HMAC appended to a ciphertext to validate that the encrypted message is unmodified (hash) and from origin (HMAC), and refuse to decrypt any message that does not pass these checks. This limits an attacker's ability to supply a ciphertext that is valid to the decryptor.

HASHLIB does not provide a construction for authenticated encryption/decryption out-of-the-box given that various use cases will have different implementations. Instead, a series of rules for properly constructing an authenticated scheme will be in the documentation for the library (*HASHLIB Quick Reference, Section 5.2*). It will be on the user to follow the rules properly, or their implementation may be insecure.

4 Conclusion

In summation, HASHLIB is a work-in-progress and sees regular updates that add resistance to various forms of attack. While its algorithmic security is solid if implemented properly, hardware-based attack is harder to defend against due to the nature of the hardware. Thus far some significant actions have been taken to secure the library against hardware-based attack and HASHLIB's code is publicly available on forums frequented by others familiar with the CE's hardware and is under continuous peer-review with the aim of continually improving the library's security profile.

If you have any questions about the proper usage of HASHLIB in your programs, any of its functionality, algorithms, etc, please do not hesitate to contact me on Discord at `acagliano#3685`. I would rather you ask questions than implement security improperly and risk compromising your implementation.

Please update to new stable releases of HASHLIB as soon as they become available and update any software you are developing to use the latest versions of HASHLIB as soon as you are able. At this stage in HASHLIB's development, updates are security enhancements, not major feature additions.

5 References

- [1] <https://github.com/B-Con/crypto-algorithms>
source of AES and SHA-256 initial algorithms ported to platform
SHA-256 later rewritten in ez80 Assembly by beekadamtheinventor

- [2] <https://www.cemetech.net/forum/viewtopic.php?p=293090#293090>
source of information on randomness on the TI-84+ CE

- [3] https://en.wikipedia.org/wiki/Cryptographically-secure_pseudorandom_number_generator
information on constructing a secure PRNG

- [5] <https://datatracker.ietf.org/doc/html/rfc8017>
guidance on implementing RSA-OAEP and RSA-PSS padding schemes

- [6] <https://datatracker.ietf.org/doc/html/rfc4868>
guidance on implementing SHA-256 HMAC