On the Methods and Applications Of Cryptography On the Texas Instruments TI-84+ CE Graphing Calculator

A Whitepaper by Anthony Cagliano

Contents

1 Introduction

`HASHLIB` was a project I began while developing another cutting-edge networking project, *VAPOR*, to provide on-device hashing capabilities. The primary function of these hashes was to allow the client, a TI-84+ CE graphing calculator, to validate the completeness of file transfers. It provided standard additive 24-bit and 32-bit checksums, CRC-32, SHA-1, and SHA-256.

It wasn't long before I realized that I could use `HASHLIB` as an opportunity to bring some more advanced cryptography to the TI-84+ CE. Thus, I endeavored to expand the capabilities of the library beyond just hashing. In its current state the library computes SHA-256 (with the simpler, non-cryptographic hashes removed) and adds the AES-CBC encryption standard, a crypto-safe random number generator, and there are plans to add the public key side of RSA (to a max key size of 2048) as well in the near future.

Some of you may be thinking "Why"? "Why do all this work on a calculator when much more secure and faster alternatives exist to implement this on?". The answer is multi-faceted. First, because I can. That is really the only answer needed. Beyond that, I wanted to prove that at least some of this could be done, reasonably fast on a TI-84+ CE. I also wanted to understand the logic behind the algorithms, so what better way to do that then to write them myself, or to derive an existing algorithm and adapt it for this device. As a consequence of undertaking this project, I now have an improved understanding of how cryptography works, as well as an understanding and respect for the professionals who do consistent and much more intricate work on much more complicated functions than I implemented to ensure that our Internet experience remains as safe as possible.

Also, I am fully aware of the general rule of using cryptography: "Don't implement your own crypto; Use a library.". However, that luxury does not exist for this platform, as no such library exists for the TI-84+ CE until HASHLIB. Therefore, to be as closely in accordance with that rule as possible, I endeavored to locate existing and tested implementations for cryptographic algorithms that were most compatible with this platform such that they would need minimal modification to work on this device. SHA-256, AES, and Base64 were all derived in this fashion. Think of those implementations more as a `porting an existing implementation` rather than a from-scratch implementation. However, the CSPRNG was written entirely from scratch as its source of entropy would need to be derived differently. In this case, I researched how to properly construct one, and the tests such a generator would need to pass to be secure. More on that later.

In this whitepaper, I will elaborate, where needed, on the methodologies I used to arrive at the algorithm I implemented, and where relevant, provide citations of routines I derived and how I modified them for this platform. I will also provide use-case examples for various algorithms, or combinations, of algorithms. I will conclude this introduction and move onto the good stuff by thanking those who have downloaded, assisted with, contributed to, or encouraged this project, and will credit contributors later in the document.

2 Abridged Library Information

2.1 Secure Pseudo-Random Number Generator

`HASHLIB` contains a secure random number generator designed to seek out the best source of entropy on the device running the code, and use that state to populate an entropy pool. See the Proofs & Extended Information section for specifics about the implementation.

Functions

hashlib_SPRNGInit(void)

Initializes the Secure PRNG (SPRNG) to read from the byte with the best entropy on the device. Adds initial entropy as well. Returns *NULL* if failure.

hashlib_SPRNGAddEntropy(void)

Reads 192 bytes from the selected byte to the internal entropy pool. Returns 0 if no source byte is set.

hashlib_SPRNGRandom(void)

Returns a pseudo random 32-bit integer derived from a hash of the entropy pool and 4 reads from the selected byte. Adds entropy as well. If no source byte is set, `hashlib_SPRNGInit()` is called a maximum of 5 times until an acceptable source byte is located. If no acceptable source byte could be found, the function returns 0.

hashlib_RandomBytes(uint8_t *ptr, size_t len)

An implementation based on *urandom*; Fills a pointed buffer *ptr* to size *len* with random numbers. Uses internal SPRNG. Because it invokes `hashlib_SPRNGRandom()`, calling this function will initialize the PRNG if it has not already been done. In this way, the PRNG is secure against "End-User Forgetfulness Attacks" (yea, I know this isn't a real "attack". But it should be.).

2.2 SHA-256

This library also contains the SHA-256 cryptographic hash. This implementation was derived from B-Con's `crypto-algorithms`[1] repository, which is in the public domain. To adapt this code to the

[1] `https://github.com/B-Con/crypto-algorithms/`

TI-84+ CE, I needed to redefine the data type "WORD" as a uint32_t, so that it would fit within the architecture of the ez80 ALU. In addition, I needed to write handler functions for resetting and adding uint64_t-type values, since neither the CE operating system nor the C toolchain contain support for this. The uint64_t handler functions were written in ez80 Assembly by `beckadamtheinventor`[2].

Functions

hashlib_Sha256Init(sha256_ctx *ctx)
Initializes the state of pointed SHA-256 context structure *ctx*.

hashlib_Sha256Update(sha256_ctx *ctx, const uint8_t *buf, size_t len)
Updates the state of the SHA-256 context *ctx* with the data at pointed buffer *buf* for *len* bytes.

hashlib_Sha256Final(sha256_ctx *ctx, const uint8_t *digest)
Finalizes the state of the SHA-256 context *ctx*, returning the hash digest into the buffer *digest*.

2.3 Advanced Encryption Standard (AES-CBC, CBC-MAC)

In addition to the faster cryptographic hashes, this library also provides two encryption implementations, the first of which is the `AES symmetric block cipher`. It accepts keys of length 128, 192, or 256 bits. This code was also derived in large part from B-Con's crypto-algorithms. However, the code on the repository did not implement plaintext padding, requiring me to write my own. That function will be elaborated on in the Miscellaneous section of this function list. In addition, I wrote functions to generate and verify a MAC for a given message. These functions implement a derivative of the IPsec standard, where you have two keys, {`k1, k2`}, and a random IV. First, the message is padded according to a specified padding scheme (see the padding function). Then the padded message is encrypted using AES-CBC over $k1$ and the random IV. Then, the IV and ciphertext is encrypted again using CBC-MAC over $k2$ with a constant IV to generate the MAC. The output message is $[IV, ciphertext, MAC(IV + ciphertext)]$.

Functions

hashlib_AESLoadKey(const uint8_t *key, aes_ctx *ks, size_t keysize)
Loads the supplied *key* of length *keysize* into the AES key schedule structure *ks*. Keysize

[2] `https://github.com/beckadamtheinventor`

is in bits, not bytes.

hashlib_AESEncrypt(const uint8_t *plaintext, size_t len, uint8_t *ciphertext, aes_ctx *ks, const uint8_t *iv)
Encrypts the indicated size *len* at pointed *plaintext* buffer, using key *ks* and initialization vector *iv*, writing the output to pointed *ciphertext*.

hashlib_AESDecrypt(const uint8_t *ciphertext, size_t len, uint8_t *plaintext, aes_ctx *ks, const uint8_t *iv)
Decrypts the indicated size *len* at pointed *ciphertext* buffer, using key *ks* and initialization vector *iv*, writing the output to pointed *plaintext*.

hashlib_AESOutputMAC(const uint8_t *plaintext, size_t len, uint8_t *mac, aes_ctx *ks)
Returns a Message Authentication Code (MAC) for the given *plaintext* into the pointed *mac* buffer for the given key schedule *ks*. This key schedule **MUST** differ from the one used to encrypt or your implementation will be insecure.

hashlib_AESEncryptWithMAC(const uint8_t *plaintext, size_t len, uint8_t *ciphertext, aes_ctx *ks_encrypt, aes_ctx *ks_mac, uint8_t pad_schm, uint8_t *iv)
Performs an AES encryption of the given *plaintext* over the key *ks_encrypt* and the initialization vector *iv*, padding the plaintext according to the scheme specified in *pad_schm*. The IV used for that as well as the ciphertext are output to the *ciphertext* buffer. Then runs AES encryption in MAC-generation mode over the key *ks_mac* for the IV and the ciphertext. The MAC is appended to the ciphertext to produce the full output ciphertext. This is based on the IPsec standard. Provided you use unique key schedules (initalized with AESLoadKey) for the encryption and the MAC generation, this scheme should offer resistance against both CPA and CCA attacks.

hashlib_AESVerifyMAC(const uint8_t *ciphertext, size_t len, aes_ctx *ks_mac)
Verifies the integrity of the given *ciphertext* of size *len* by running all but the last block through AES encryption for key *ks_mac* to generate a MAC. That MAC is then compared with the last block of the ciphertext. A match returns True, no match returns False. This function expects the IPsec standard.

2.4 Base64 Encode/Decode

As part of my effort to implement **bcrypt**, I decided to implement Base64 encoding and decoding. I started off writing these myself, but during the arduous process of attempting to debug bcrypt, I wound up scrapping my own routines for ones from the `OpenBSD repository`[3].

Functions

hashlib_b64encode(char *b64buffer, const uint8_t *data, size_t len)
Base64-encodes *len* bytes at buffer *data* and writes the output to $b64buffer$.

hashlib_b64decode(uint8_t *buffer, size_t len, const char *b64data)
Base64-decodes *len* bytes at buffer $b64data$ and writes the output to $buffer$.

2.5 RSA

The `RSA` implemention planned for this library will be one of my own design with minimum derivation from external code. It will also cover only the public key side of the protocol, as generating a key pair on the calculator would be painfully slow. The implementation will also depend on beckadamtheinventor's planned *big-int* library. It will support variable key sizes from 512 bits to 1024 bits. As the implementation is still work-in-progress, that's really all I can say about it at the time.

2.6 Miscellaneous

Lastly, there are a few functions that exist in this library, dissociated from specific implementations but useful for other purposes. In this section, you will see some information on them.

hashlib_EraseContext(void *ctx, size_t len)
Fills the given context at *ctx* with zeroes for *len* bytes to destroy any residual cryptographic state information in those contexts. It is recommended to call this as soon as you are finished with any cryptographic implementation that leaves state info in memory.

hashlib_PadInputPlaintext(const uint8_t *plaintext, size_t len, uint8_t *outbuf, uint8_t alg, uint8_t schm)
Pads the input *plaintext* according to the selected scheme *schm* and algorithm *alg* and writes the padded output to $outbuf$. It is recommended to just pass `SCHM_DEFAULT` as

[3] `https://github.com/openbsd/src/blob/master/lib/libc/crypt/bcrypt.c`

the padding scheme. This selects PKCS7 for AES-CBC and AES-CBC-MAC and OAEP encoding for RSA. The available padding modes are:

(a) (AES) SCHM_PKCS7 (pad with the length of the padding)

(b) (AES) SCHM_ISO_M2 (pad with the bytes 0x80 followed by 0x00 for the remaining length)

(c) (AES) SCHM_ANSIX923 (pad with pseudorandom bytes to the remaining length)

(d) (RSA) SCHM_RSA_OAEP (pad to constant 256-byte lenth like so: pad message to 240 bytes and generate 16-byte salt. Apply Feistel network by SHA-256 hashing the salt and then cyclically xor'ing that with the message, then SHA-256 hashing the encoded message and xor'ing that with the salt (truncating to 16 bytes)).

hashlib_StripPadding(const uint8_t* plaintext, size_t len, uint8_t* outbuf, uint8_t alg, uint8_t schm);)

Strips the padding from the given *plaintext* for the given padding scheme *schm* and algorithm *alg* and returns the size of the unpadded message in the return value as well as the unpadded message in *outbuf*. However, for the `SCHM_ANSIX923` scheme, it merely returns the input size. Due to the randomness of that scheme, there is no way to determine the length of the padding; The user would have to already know it.

hashlib_CompareDigest(const uint8_t* digest1, uint8_t* digest2, size_t len);

Compares *len* bytes at the buffer *digest*1 with *len* bytes at the buffer *digest*2. This is provided here as an alternative to the C standard `memcmp` or string comparison functions which return immediately upon encountering a value i where $digest1[i] != digest2[i]$. This makes those routines vulnerable to timing attacks. This function covers for that vulnerability by ensuring that the entire buffer is always compared regardless of where (or if) such a value i is found. Special thanks to the `Cemetech` user `jacobly`[4] who provided an ez80 Assembly version of this routine to fix the C version which was still vulnerable to this attack due to compiler optimization.

[4] `https://www.cemetech.net/forum/profile.php?mode=viewprofile&u=5162`

3 Proofs & Extended Information

In this section, implementations that I devised myself will have their mechanisms elaborated on and, where applicable, mathematical proofs show. Please note that, in accordance with the general rule of thumb of "don't write your own crypto", I used adapted versions of tested algorithms where possible.

3.1 SPRNG Elaboration

The `CSPRNG` is the most platform-dependent part of this code, and consequentially the part of it I did the most work 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`[5] 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 led me to write an algorithm, (with the help of `beckadamtheinventor`, who converted my C code into faster ez80 Assembly) 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 `CSPRNG` 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 `CSPRNG` functions will return either `0` or `NULL`.

The internal state of the `CSPRNG` also reserves an entropy pool 192 bytes in length to which it reads from the optimal byte 192 times every time a random number is generated or an Add Entropy function is called by the user.

To generate a random number, the optimal byte is read from 4 times to produce an initial state of low entropy for the random 32-bit number. After this, the entropy pool is hashed using SHA-256 to generate a 32-byte digest of high entropy. That hash is broken into 4-byte blocks (8 iterations), with bytes 1-4 of each hash iteration being xor'ed with bytes 1-4 of the random number, sequentially. The `Add Entropy` function is called at this point to scramble the state, even if the user does not call `Add Entropy` themselves. This ensures that this algorithm passes the next-bit test and state compromise tests, as even with moderate bias, and full knowledge of the current state of the entropy pool, it is nearly impossible to predict the result of 192 reads that would affect the next generated number with greater than 50% probability.

[5] `https://www.cemetech.net/forum/profile.php?mode=viewprofile&u=29638`

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% (± 25% deviation from 50/50), with the rest of the byte likely having a much higher bias. To this end, I 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:

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

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

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

Multiplying the entropy for a byte in the entropy pool (pending data on correlation) by the 192 bytes of the entropy pool plus the initial 4-byte read to the rand, you arrive at:

$$H(x)_{min}(192 + 4) = 0.811 \times 196.0 = 159.1$$

This means we have 159.1 bits of entropy per 32-bit number at minimum. Because the available entropy exceeds (by a lot) the bit-width of the number we are generating, I assert that the probability of `r <-- {0,1}` is within a negligible deviation, ε, from $\frac{1}{2}$ for all bits in the 32-bit number.

Let A be an algorithm to test probability bit b is set s.t.$A : A(b) = (b == 1)$

$$Adv(PRG) = [\Pr(A(G(x))) - \Pr(r \leftarrow \{0,1\})] \forall_b \leftarrow \{0,1\}^{32}$$

$$Adv(PRG) = [(\frac{1}{2} \pm \varepsilon) - \frac{1}{2}]$$

$$Adv(PRG) = \pm\varepsilon$$

Because the advantage is negligible, the output of the CSPRNG is not distinguishable from a uniform distribution over the same 32 bits, thus this PRNG is secure. Bear in mind that this is with **maximum allowable bias**; The chosen entropy source will likely have a lower bias, and thus the entropy will become even higher and the advantage even closer to zero.

4 Applications for HASHLIB

Surprisingly, HASHLIB can have a few practical applications other than making its developer wish he was dead (which was a quite frequent occurrence during the development process). In fact, this entire project was born out of the need for a cryptographic hash to validate the completeness of a file transfer, and as such, this is a good example of a use for it. In addition, AES is one of, if not the most, secure block ciphers in usage today. Not to mention, it actually runs faster than the Blowfish implementation that I had put into a previous release of this library that is now removed.

From a practical standpoint, you should not be doing things on a calculator that require decent security, like logging into secure servers, sending credit card info, and more anyway. Whatever cryptographic mechanisms I can pull off on a graphing calculator with a 48 mHz CPU is likely child's play compared to what a computer leveraging a CPU orders of magnitude faster can do. It would follow that I would need to choose between implementing strict security resulting in an algorithm that takes a year and a half to run, or sacrifice a little bit of security for a little bit of speed. Where possible, I chose the most secure mechanism I could.

This library offers two encryption implementations–one symmetric block cipher (AES) and one asymmetric public key cipher (RSA). The RSA implementation is encryption-only. It is intended to be used in a server-centric setup where, upon connection to a remote host, the server generates a public and private key pair and shares the public key with the client. The client, in this case, your calculator, generates an AES key in the length of 128-256 bits. Longer keys are more secure but will cause slower encryption. This key should be RSA encrypted with the public key and then sent back to the remote host. With both the client and the remote host now in possession of a shared secret, AES can be used for symmetric encryption when data security is needed. As these implementation will slow down your data transfers noticeably, it is recommended to only use them when needed. As previously mentioned, if you need genuine security, please do not use a calculator for what you are doing. I intended this library as proof of concept, a learning exercise for myself, and something fun for the TI programming community to experiment with.

5 References

[1] B-Cons Crypto Algorithms

[2] Cemetech — Producing Crypto-Safe Randomness on the TI-84+ CE

[3] Wikipedia — CSPRNG Information

[5] Wikipedia — Base64 Implementation

6 Contributors

[1] beckadamtheinventor [coding assistance]

[2] commandblockguy [coding assistance]

[3] Zeroko [information on entropy on TI-84+ CE]

[3] jacobly [ez80 Assembly port of hashlib_CompareDigest()]