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

A Whitepaper by Anthony Cagliano

Contents

1 Library Quick Reference **2**

1.1 Secure Pseudo-Random Number Generator 2

1.2 Cryptographic Hashes 2

1.3 Advanced Encryption Standard (AES) 3

1.4 RSA 5

1.5 Miscellaneous 7

2 HASHLIB Cryptanalysis **9**

2.1 SPRNG Elaboration 9

2.2 Symmetric Encryption – AES 11

2.3 Asymmetric Encryption – RSA 11

3 Applications for HASHLIB **12**

3.1 Recommended Secure Infrastructure 12

3.2 Example Code for Secure Session 13

3.3 Need Help? 15

4 References **16**

5 Contributors **16**

6 Disclaimer & Legal Notice **16**

1 Library Quick Reference

1.1 Secure Pseudo-Random Number Generator

void* 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.

uint32_t hashlib_SPRNGRandom(void)

Returns a psuedo-random 32-bit integer derived from a hash of the entropy pool. Adds entropy before and after the random generation 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.

bool hashlib_RandomBytes(uint8_t *ptr, size_t len)

A function to generate an arbitrary number of bytes. Based on */dev/urandom*.

ptr Pointer to a buffer to write random bytes to.

len Number of random bytes to write.

1.2 Cryptographic Hashes

void hashlib_Sha256Init(sha256_ctx *ctx, uint32_t *mbuffer)

Initializes the state of a SHA-256 context.

ctx Pointer to a SHA-256 context structure.

mbuffer Pointer to a scratch buffer for the algorithm to use.

```
uint32_t mbuffer[SHA256_MBUFFER_LEN];
```

void hashlib_Sha256Update(sha256_ctx *ctx, uint8_t *buf, size_t len)

Updates the state of the SHA-256 context with the data at the passed buffer.

ctx Pointer to a SHA-256 context structure.

buf Pointer to a buffer containing data to add to the hash state.

len Length of data at **buf** to hash.

void hashlib_Sha256Final(sha256_ctx *ctx, const uint8_t *digest)

Finalizes the state of the SHA-256 context *ctx*, returning the hash digest.

ctx Pointer to a SHA-256 context structure.

digest Pointer to a buffer to write the hash digest to. Must be at least 32 bytes.

void hashlib_MGF1Hash(const uint8_t *data, size_t len, uint8_t *outbuf, size_t outlen)

Returns an MGF1 hash of arbitrary length using the SHA-256 algorithm.

data Pointer to data to hash.

len Length of data at **data** to hash.

outbuf Pointer to buffer to write MGF1 hash to.

outlen Number of bytes of MGF1 hash to write.

outbuf must be at least **outlen** bytes large.

1.3 Advanced Encryption Standard (AES)

bool hashlib_AESLoadKey(const uint8_t *key, aes_ctx *ks, size_t keysize)

Loads the bytearray representing the AES key into an AES key schedule context.

key Bytearray containing AES key.

ks Pointer to an AES key schedule context.

keysize Length of AES key, in bytes.

bool hashlib_AESEncryptBlock(const uint8_t* block_in, uint8_t* block_out, const aes_ctx* ks)

Encrypts a single block (16 bytes) of data using AES, in ECB cipher mode.

block_in Block of data to encrypt.

block_out Buffer to write encrypted data to.

ks AES key schedule context.

block_in and block_out are aliasable.

ECB mode is insecure. This is exposed as a constructor for other cipher modes by experienced users – use CBC or CTR modes unless you know what you are doing.

bool hashlib_AESDecryptBlock(const uint8_t* block_in, uint8_t* block_out, const aes_ctx* ks)

Decrypts a single block (16 bytes) of data using AES, in ECB cipher mode.

block_in Block of data to decrypt.

block_out Buffer to write decrypted data to.

ks AES key schedule context.

block_in and block_out are aliasable.

ECB mode is insecure. This is exposed as a constructor for other cipher modes by experienced users – use CBC or CTR modes unless you know what you are doing.

hashlib_AESEncrypt(const uint8_t *plaintext, size_t len, uint8_t *ciphertext, aes_ctx *ks, const uint8_t *iv, uint8_t ciphermode)

General-purpose AES encryption.

plaintext Pointer to data to encrypt.

len Size of the data at **plaintext**.

ciphertext Buffer to write encrypted data to.

ks AES key schedule context.

iv Initialization vector (16-byte random bytearray).

ciphermode The cipher mode to use. Options: CBC, CTR

plaintext and ciphertext are aliasable.

hashlib_AESDecrypt(const uint8_t *ciphertext, size_t len, uint8_t *plaintext, aes_ctx *ks, const uint8_t *iv, uint8_t ciphermode)

General-purpose AES decryption.

ciphertext Pointer to data to decrypt.

len Size of the data at **ciphertext**.

plaintext Buffer to write encrypted data to.

ks AES key schedule context.

iv Initialization vector (16-byte random bytearray).

ciphermode The cipher mode to use. Options: CBC, CTR

plaintext and ciphertext are aliasable.

hashlib_AESOutputMac(const uint8_t *buf, size_t len, uint8_t *mac, aes_ctx *ks)

Returns a Message Authentication Code (MAC) for the data at the given buffer using a distinct AES key schedule.

buf Pointer to data to generate MAC for.

len Size of the data at **buf**.

mac Buffer to write MAC to.

ks AES key schedule context for MAC.

> Do not use the same key schedule you use for encryption or your implementation will be insecure.

hashlib_AESPadMessage(const uint8_t *plaintext, size_t len, uint8_t *outbuf, uint8_t schm)

Pads the plaintext according to the indicated padding scheme.

plaintext Pointer to data to pad for AES encryption.

len Size of the data at **plaintext**.

outbuf Buffer to write padded data to.

schm Padding scheme to use. Options: SCHM_PKCS7, SCHM_ISO_M2, SCHM_DEFAULT

> It is recommended to pass `SCHM_DEFAULT` as the padding scheme.

hashlib_AESStripPadding(const uint8_t *plaintext, size_t len, uint8_t *outbuf, uint8_t schm)

Strips the padding from the plaintext according to the indicated padding scheme.

plaintext Pointer to data to remove padding from.

len Size of the data at **plaintext**.

outbuf Buffer to write unpadded data to.

schm Padding scheme to use. Options: SCHM_PKCS7, SCHM_ISO_M2, SCHM_DEFAULT

1.4 RSA

hashlib_RSAEncodeOAEP(const uint8_t *plaintext, size_t len, uint8_t *outbuf, uint8_t modulus_len)

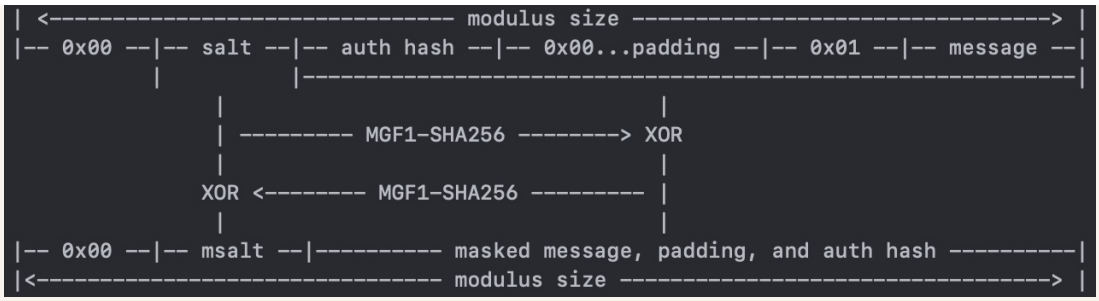
Pads the RSA plaintext to the modulus length according to the Optimal Asymmetric Encryption Padding (OAEP) encoding scheme.

plaintext Pointer to data to pad for RSA encryption.

len Size of the data at **plaintext**.

outbuf Buffer to write padded data to.

modulus_len The length of the RSA key to pad for.



hashlib_RSADecodeOAEP(const uint8_t *plaintext, size_t len, uint8_t *outbuf)

Reverses the OAEP encoding scheme detailed in `RSAEncodeOAEP()` on the plaintext.

plaintext Pointer to OAEP-encoded data to decode.

len Size of the data at **plaintext**.

outbuf Buffer to write decoded data to.

hashlib_RSAEncodePSS(const uint8_t *plaintext, size_t len, uint8_t *outbuf, uint8_t modulus_len, uint8_t *salt)

Pads the RSA plaintext according to the Probabilistic Signature Scheme (PSS).

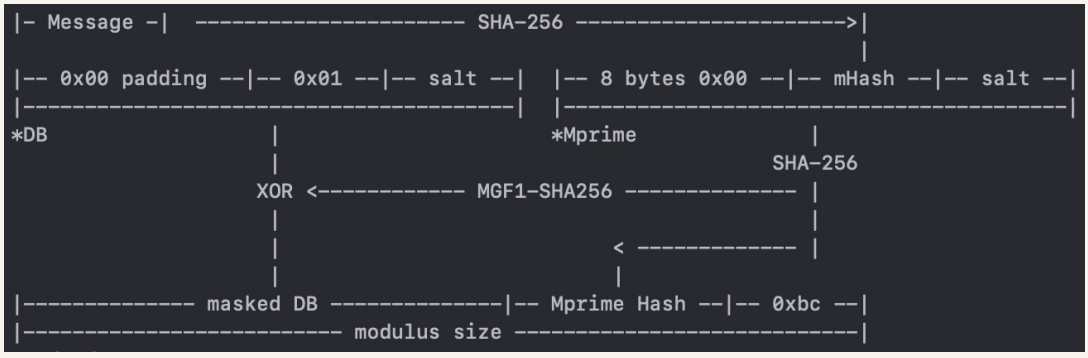
plaintext Pointer to data to pad for RSA encryption.

len Size of the data at **plaintext**.

outbuf Buffer to write padded data to.

modulus_len The length of the RSA key to pad for.

salt A string of random bytes equal in length to the SHA-256 hash digest.



hashlib_RSAEncrypt(const uint8_t *msg, size_t msglen, uint8_t *ciphertext uint8_t *pubkey, size_t keylen)

Encrypts a plaintext given a modulus (RSA key) and the public exponent 65,537.

msg Pointer to data to encrypt using RSA.

msglen Size of the data at **msg**.

ciphertext A buffer to write the output ciphertext to. Must be at least **keylen** bytes large.

pubkey Public encryption modulus (RSA key).

keylen The length of the RSA key (public modulus).

> This function is resistant to timing analysis if arguments are in non-accelerated RAM.
>
> Arguments are bytearrays, and are therefore non-endian.
>
> **msglen** must equal **keylen**.

> This algorithm will automatically apply OAEP encoding to the plaintext prior to encryption. You should not need to call `hashlib_RSAEncodeOAEP` yourself for general use.

hashlib_SSLVerifySignature(const uint8_t *ca_pubkey, size_t keysize, const uint8_t *cert, size_t certlen, uint8_t sig_alg)

Attempts to verify the SSL certificate specified using the certifying authority's public key specified.

ca_pubkey Pointer to the certifying authority's public key.

keysize Size of public key.

cert Pointer to the certificate to verify.

certlen Length of the certificate.

> HASHLIB cannot authenticate the metadata in the certificate with the CA as a browser would before assuming the host is legitimate. It can only perform signature validation. You will need to pair with another protocol, such as a custom server, to verify certificates.

1.5 Miscellaneous

hashlib_EraseContext(void *ctx, size_t len)

Writes zeroes to the indicated number of bytes starting at the given address.

ctx Pointer to the data to zero.

len Number of bytes to zero.

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

Compares the given number of bytes in the two bytearrays supplied.

digest1 Pointer to the first string of bytes to compare.

digest2 Pointer to the second string of bytes to compare.

len Number of bytes to compare.

This function is resistant to timing analysis.

hashlib_ReverseEndianness(const uint8_t* in, uint8_t* out, size_t len)

Reverses the byte order of the given bytearray of given length.

in Pointer to the bytearray to reverse.

out Pointer to the buffer to write output.

len Number of bytes in **in**.

in and **out** are not aliasable.

2 HASHLIB Cryptanalysis

HASHLIB implements some of the most secure encryption in use today – namely AES and RSA. The library provides all the functions need to ensure plaintext obfuscation and ciphertext integrity between your calculator and a remote host. Using HASHLIB, it is possible to handshake on an RSA public key with a remote host, transmit an AES session key to the remote host encrypted with that public key, and then communicate securely over AES. In the following subsections I will provide cryptanalysis of the main implementations HASHLIB provides.

2.1 SPRNG Elaboration

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 ± 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 internal state of the `SPRNG` also reserves an entropy pool 119 bytes in length. Each byte in the entropy pool is updated using a composite of seven (7) distinct reads xored together.

To generate a random number, first the entropy pool is updated in overwrite mode, not xor mode to ensure that the SPRNG state before a random generation cannot leak any information about the next random 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 four (4) 8-byte blocks that are xored together. Those four blocks are the random uint32_t. The entropy pool is updated again in overwrite mode so that the SPRNG state after the generation does not leak any information about the last random number generated. This ensures that this algorithm passes state compromise tests. Also I assert that this generator passes the next-bit test as well, as hashing a bit-stream of sufficient entropy produces output indistinguishable from randomness. In addition, the SPRNG passes all statistical tests regardless of the sample size.

[1] `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, 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:

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

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 by the 119 bytes of the entropy pool you arrive at:

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

This means we have 96.51 bits of entropy per 32-bit number at minimum. Because the available entropy exceeds (by more than double) 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. 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.

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. To combat this, the `hashlib_SPRNGAddEntropy()` function was rewritten with direction from Zeroko. The size of the entropy pool was reduced from 192 bytes to 119 bytes (to fit the SHA compression within 2 blocks for speed), but each byte in the entropy pool is a composite byte attained by xor'ing seven (7) distinct reads from the optimal address together. This serves to ensure that, even with correlation, the total entropy of the system is fairly close to the calculated 96.51 bits. The code alterations also cause the SPRNG to run about 1.5 times faster than before.

2.2 Symmetric Encryption – AES

The included AES implementation provides a front-end for using CBC and CTR modes, which are both commonly used in secure network infrastructures. It is derived in large part from `B-Con's crypto-algorithms`[2], which is a public domain repository of cryptographic implementations constrained within 32-bit data types. The ECB mode single-block functions are also exposed for use by those knowledgeable in cryptography to construct their own cipher modes, but most users of this library can safely use the provided `hashlib_AESEncrypt()` and `hashlib_AESDecrypt()` functions. Using a secure key and secure IVs generated by the library's SPRNG will provide plaintext security, and using the provided SHA-256 or CBC-MAC authentication mechanisms will provide ciphertext integrity. As of now, the AES implementation has an unknown level of resistance to timing analysis and side-channel attack.

2.3 Asymmetric Encryption – RSA

The included RSA implementation provides not only encryption but also the OAEP and PSS padding schemes. The RSA encryptor will reject an unpadded plaintext to ensure that the implementation does not expose itself to any mathematical weaknesses. The embedded modular exponentiation function, written by CE C toolchain coauthor `jacobly`[3], leaks the modulus size and public exponent but not the modulus (key) itself via timing analysis, meaning that the RSA implementation is resistant to that type of attack so long as the buffers passed are in non-accelerated memory. In addition, OAEP padding adds plaintext security. End users can also add ciphertext integrity by appending a hash or CBC-MAC to the encrypted message. The RSA implementation has an unknown level of resistance to side-channel attack.

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

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

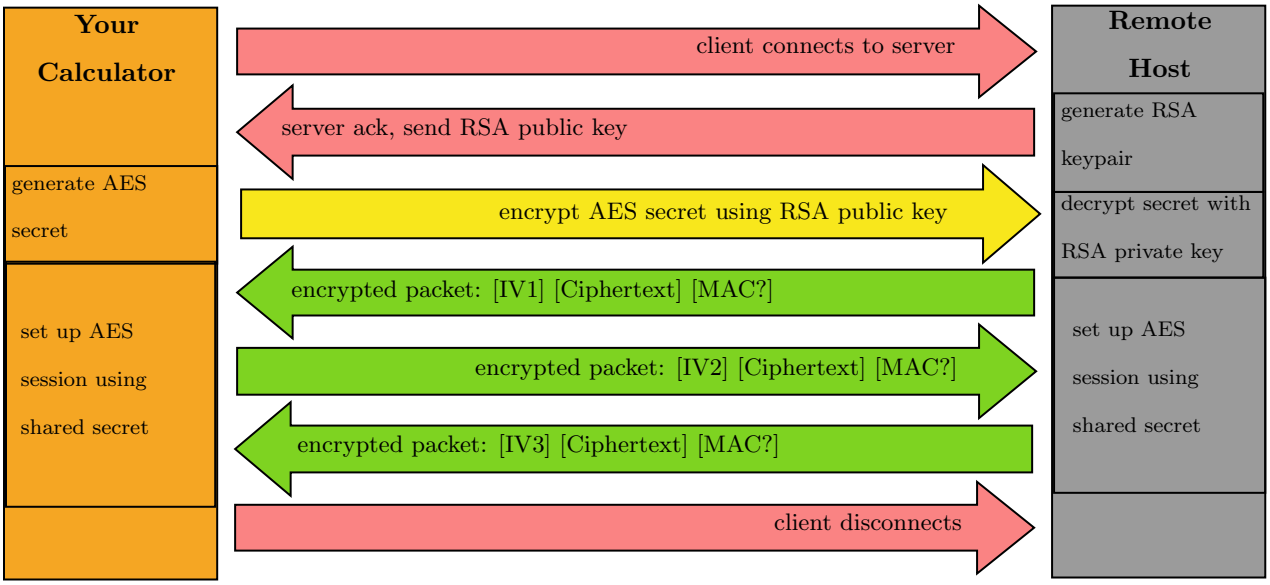
3 Applications for HASHLIB

Surprisingly, HASHLIB can have a few practical applications other than giving its developer an early run-in with high blood pressure. 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 probably the most secure block cipher in usage today. Not to mention, it actually runs faster than the Blowfish implementation that was in a previous release of this library.

HASHLIB is a composite of several ported industry-standard forms of cryptography, so it is fair to assert that HASHLIB is secure as long as the SPRNG generating the keys is also secure (which is proved in an earlier section). However, do not take this to mean that I advocate the transmission of ultra-sensitive information like credit card information and online banking/account passwords from your calculator. There are attack vectors against the TI-84+ CE that this library cannot protect against, which I described in the previous *Caveats* section. That being said, it is perfectly ok, and probably safe, to use HASHLIB to encrypt things like logging into online calculator game accounts (i.e. Project TI-Trek), downloading software (i.e. Project Vapor, BOS Package Manager), and other similar things. This section will elaborate on the suggested way in which to initiate a secure session with HASHLIB, as well as provide some C code samples to demonstrate how to implement it.

3.1 Recommended Secure Infrastructure

In the graphic below, red arrows indicate unencrypted packets, yellow indicates a packet encrypted using an RSA public key, and green indicates an AES session.



3.2 Example Code for Secure Session

In this section you will see example code from another one of my projects, `Project TI-Trek`, that I used to construct a secure session for encrypting a login token for the game. Notes will be scattered throughout to help you port the example into functional code for your own usage.

Listing 1: Client connects to remote host

```
    /*
      ntwk_send() and ntwk_send_nodata() are variadic functions
      that construct a packet from their inputs
    */
    ntwk_send(CONNECT, PS_STR(<pointer to string containing IP>))

```

Listing 2: Server acknowledges client and sends public key

```
    def init_secure_session(self):
      try:
        self.rsa_key = RSA.generate(1024)
        pubkey_bytes = bytes(self.rsa_key.publickey().exportKey('DER'))[29:29+128]
        self.send([ControlCodes["REQ_SECURE_SESSION"]] + list(pubkey_bytes))
        return
      except: self.elog(traceback.format_exc(limit=None, chain=True))

```

Listing 3: Client generates AES session key and sends to server

```
  #define AES_KEYLEN 32
  // data = pointer pubkey sent by server
  case REQ_SECURE_SESSION:
  {
    uint8_t aes_key[AES_KEYLEN];
    uint8_t msg[RSA_PUBKEY_LEN] = {0};
    gfx_TextClearBG("generating AES key...", 20, 190);
    hashlib_RandomBytes(aes_key, AES_KEYLEN);
    hashlib_RSAEncodeOAEP(aes_key, AES_KEYLEN, msg, RSA_PUBKEY_LEN, NULL);
    gfx_TextClearBG("RSA-1024 encrypting...", 20, 190);
    hashlib_RSAEncrypt(msg, RSA_PUBKEY_LEN, data, RSA_PUBKEY_LEN);
    ntwk_send(RSA_SEND_SESSION_KEY, PS_PTR(msg, RSA_PUBKEY_LEN));
    break;
  }

```

Listing 4: Server decrypts AES session key and sends ack

```
    def setup_aes_session(self, data):
```

```
        try:
          cipher = PKCS1_OAEP.new(self.rsa_key, hashAlgo=SHA256)
          self.aes_key = cipher.decrypt(bytes(data[1:]))
          del self.rsa_key
          self.send([ControlCodes["RSA_SEND_SESSION_KEY"]])
          return
        except: self.elog(traceback.format_exc(limit=None, chain=True))

```

Listing 5: Client encrypts login token with AES

```
    /*
      Repeat this style of code as needed for each encrypted packet needed.
      If you plan to send more than 2^(B/2) packets (where B is the bit-length of
      the AES key) using AES, you will need to cycle (change) your AES session key
      once you have sent 2^(B/2) packets.
      This will require you to use RSA to exchange
      the new key with the remote host again.
    */

    bool gui_Login(uint8_t* key) {
      aes_ctx ctx;
      uint8_t iv[AES_BLOCKSIZE];
      uint8_t ppt[PPT_LEN];
      uint8_t ct[PPT_LEN];

      gfx_TextClearBG("encrypting auth token...", 20, 190);
      hashlib_AESLoadKey(key, &ctx, AES_KEYLEN);
      hashlib_RandomBytes(iv, AES_BLOCKSIZE);

      // Pad plaintext
      hashlib_AESPadMessage(settings.login_key, LOGIN_TOKEN_SIZE, ppt, SCHM_DEFAULT);

      // Encrypt the login token with AES-256
      hashlib_AESEncrypt(ppt, PPT_LEN, ct, &ctx, iv, AES_MODE_CBC);
      gfx_TextClearBG("logging you in...", 20, 190);
      ntwk_send(LOGIN, PS_PTR(iv, AES_BLOCKSIZE), PS_PTR(ct, PPT_LEN));

      // erase the key when done to prevent state reconstruction
      hashlib_EraseContext(key, AES_KEYLEN);

      // you can also erase the IV as well
      hashlib_EraseContext(iv, AES_BLOCKSIZE;
      return;
    }

```

Listing 6: Server decrypts AES packet

```
    def log_in(self, data):
      try:
        iv = bytes(data[1:17])
        ct = bytes(data[17:])
        cipher = AES.new(self.aes_key, AES.MODE_CBC, iv=iv)
        padded_key = cipher.decrypt(ct)
        padding = padded_key[len(padded_key)-1]
        key = padded_key[0:-padding]
        hashed_key=hashlib.sha512(bytes(key)).hexdigest()
        ...
        # proceed to authenticate against saved keys

```

HASHLIB provides the secure infrastructure, but it is to the user to not break the security of the implementation by using it incorrectly. There are many rules in cryptography that cover key generation, generation and use of nonces, repetition, key cycling, authentication, and much much more. It would take more time than I have to include them all here, and doing so is not within the scope of this document. However, important notes and warnings on what not to do in are included in the function documentation. It is as easy to get cryptography wrong as it is hard to implement correctly. If you are unsure of how to use any of this, please use one of the listed methods of getting support in the **Need Help** section below.

3.3 Need Help?

Cryptography is a complex discipline and despite my best efforts to document the library, some things may still be unclear. If this is the case, I would rather you ask than risk making a mistake and compromise your effort to secure whatever you are doing. Below, you will find some of the best methods for contacting me for help, should you need to.

- EMAIL: acagliano97@gmail.com
- DISCORD: acagliano#3685
- CEMETECH: https://www.cemetech.net/forum/profile.php?mode=viewprofile&u=1779

4 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

5 Contributors

[1] ACagliano [cryptographer, lead developer]

[2] beckadamtheinventor [coding assistance]

[3] commandblockguy [coding assistance]

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

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

6 Disclaimer & Legal Notice

This software is provided free and is released in accordance with the MIT open-source license.

HASHLIB is a work-in-progress and has seen very little time as the forerunning cryptography library for the TI-84+ CE calculator. This means that it has not had much time to be thoroughly analyzed, and due to some hardware constraints may never offer total security against every possible attack. For this reason, I heavily advise that however secure HASHLIB may be, you never use it for encrypting truly sensitive data like online banking and other accounts, credit card information, and the like over an insecure network. It is

likely safe enough to be used to encrypt data transfers and account login for TI-84+ CE game servers and package managers like the ones currently under development. By using this software you release and save harmless its developer(s) from liability for data compromise that may arise should you use this software.