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 and MGF1

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]. In addition, the Mask Generation Function, MGF1 is also implemented, predominantly for use with RSA's OAEP encoding, but is also exposed for other uses as well.

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*.

hashlib_MGF1Hash(const uint8_t *data, size_t len, uint8_t *outbuf, size_t outlen)
Derives an arbitrary-length hash from the data pointed to by *data* of size *len*. Uses the SHA-256 cryptographic hash by hashing the *data* buffer with a four (4) byte big-endian counter appended for each 32-byte block of *outlen* needed. At most *outlen* bytes of output from this function are written to *outbuf*.

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

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

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 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_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.

hashlib_AESAuthEncrypt(const uint8_t *padded_plaintext, size_t len, uint8_t *ciphertext, aes_ctx *ks_encrypt, aes_ctx *ks_mac, uint8_t *iv)

Performs an AES Authenticated Encryption using the CBC-MAC standard. First the plaintext pointed to by *padded_plaintext* is encrypted using the encryption key schedule *ks_encrypt* and the initialization vector *iv*. The *iv* is written to the pointed *ciphertext* buffer and the output encrypted ciphertext is written to $ciphertext + AES\_BLOCKSIZE$. Then, the ciphertext (with the IV) is encrypted again via CBC-MAC, using the authentication key schedule *ks_mac* with a constant IV. The output of the CBC-MAC algorithm is appended to the ciphertext. The encryption and authentication key schedules must be unique. The message must be padded properly before being passed to this function, use the function provided for this.

hashlib_AESAuthDecrypt(const uint8_t *ciphertext, size_t len, uint8_t *plaintext, aes_ctx *ks_decrypt, aes_ctx *ks_mac)

Decrypts an AES ciphertext with message authentication. First, the pointed *ciphertext* (with the IV), less the last block, is encrypted via CBC-MAC using the authentication key schedule *ks_mac*. That MAC is then compared with the last block of the ciphertext. If the MAC tags do not match, the decryption fails and returns $False$. Otherwise it proceeds to decrypt the ciphertext, passing IV (first block of the ciphertext) as well as the encrypted message (second block of the ciphertext through the next to last block) and the decryption key schedule *ks_decrypt*. The resulting decrypted message is written to the pointed *plaintext* buffer. Padding will not be stripped, use the function provided for this.

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

Pads the input *plaintext* according to the selected AES scheme *schm* and writes the padded output to $outbuf$. It is recommended to just pass `SCHM_DEFAULT` as the padding scheme. The available padding modes are:

(a) SCHM_DEFAULT (selects SCHM_PKCS7)

(b) SCHM_PKCS7 (pad with the length of the padding)

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

(d) SCHM_ANSIX923 (pad with pseudorandom bytes to the remaining length)

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

Strips the padding from] the input *plaintext* according to the selected AES scheme *schm* and writes the padded output to *outbuf*. It is recommended to just pass `SCHM_DEFAULT` as the padding scheme. The padding modes are the same as above. Please note that SCHM_ANSIX923 padding cannot be stripped and the function will simply return the input size. You will need to know the expected plaintext size yourself if using that mode.

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.

hashlib_RSAPadMessage(const uint8_t* plaintext, size_t len, uint8_t* outbuf, size_t modulus_len)
Pads the input *plaintext* buffer of size *len* RSA plaintext according to the OAEP encoding scheme and writes the result to output *outbuf*. This is a Fiestel network in which: (1) The message is padded with zeros to the modulus size minus 16 bytes, (2) A 16-byte salt is returned, (2) The salt is hashed with MGF1, (3) The hashed salt is xor'ed cyclically with the padded message to create the encoded message, (4) The encoded message is hashed with MGF1, (5) The hashed encoded message is xor'ed with the salt to create the encoded salt, (6) The encoded salt is concatenated to

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

the encoded message to produce the encoded plaintext.

hashlib_RSAStripPadding(const uint8_t* plaintext, size_t len, uint8_t* outbuf)
Reverses the OAEP encoding scheme detailed in `RSAPadMessage` on the input *plaintext* buffer and writes the result to the *outbuf*.

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_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 `SPRNG` 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 `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 internal state of the `SPRNG` also reserves an entropy pool 119 bytes in length to which it reads from the optimal byte 7 times per byte (833 reads total) every time a random number is generated or `hashlib_SPRNGAddEntropy()` is called by the user.

To generate a random number, first `hashlib_SPRNGAddEntropy()` is called to scramble the state. This ensures 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 xor'd together. Those four blocks are the random uint32_t. The `hashlib_SPRNGAddEntropy()` function is called at this point to scramble the state again so that the SPRNG state does not leak any information about the last random number generated. 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. The state of the SPRNG also does not help you predict its output.

[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 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 the 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 bits to 119 bits (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.

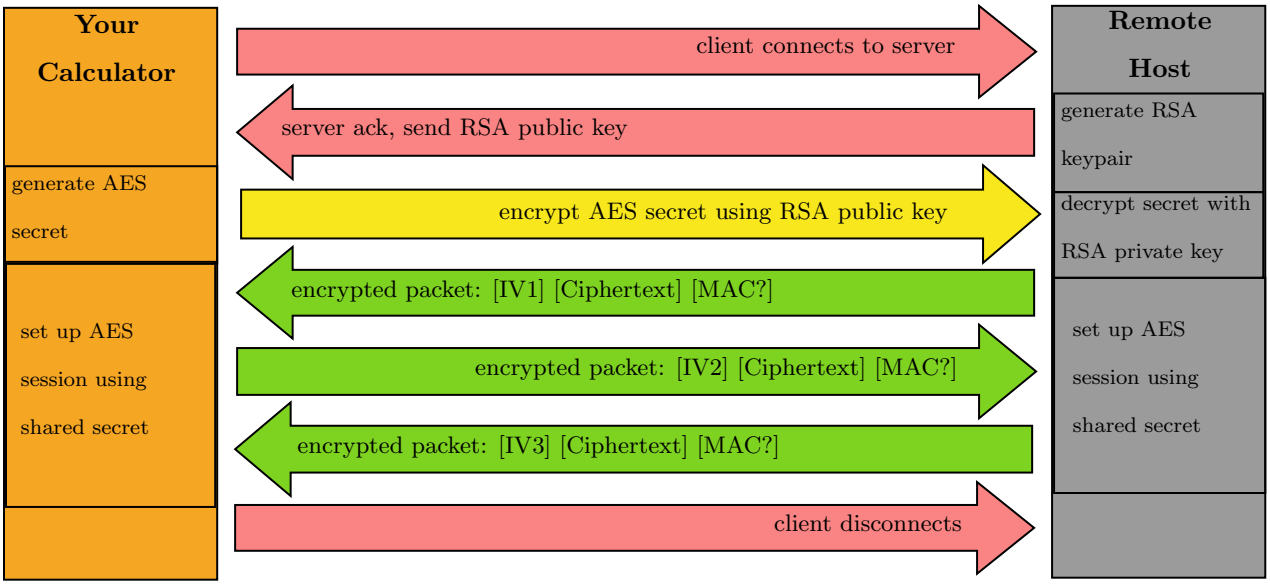
4 Applications for HASHLIB

Surprisingly, HASHLIB can have a few practical applications other 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 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.

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 I 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 caveats to using HASHLIB on your calculator, predominantly memory and speed. 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. In this section I 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.

4.1 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.



4.2 Client-Side Session Handling

There are three distinct portions of the client-side infrastructure: (1) the connection, (2) the public key encryption, and (3) the AES session. The connection section is a bit difficult to demonstrate in code because depending on your use case you may handle it differently. Thus, I will denote the sending of a packet as though it is a variadic function with the first argument being the packet type/ID and the other argument(s) being the packet itself.

Listing 1: Connect to Server

```
    ntwk_send(CONNECT, <packet>)
    // wait for server response
```

Listing 2: RSA encrypt session key

```
    // server response, Welcome message.
    // *packet, formatted [WELCOME, 2048-bit RSA public key]
    #define RSA_MOD_SIZE (2048>>3)
    #define AES_KEY_SIZE (256>>3)
    uint8_t aes_key[AES_KEY_SIZE*2];    // one for encrypt, one for MAC
    uint8_t oaep_encoded[RSA_MOD_SIZE];
    hashlib_RandomBytes(aes_key, AES_KEY_SIZE*2);
    hashlib_RSAPadMessage(aes_key, AES_KEY_SIZE*2, oaep_encoded, RSA_MOD_SIZE);
    hashlib_RSAEncrypt(oaep_encoded, RSA_MOD_SIZE, &packet[1], RSA_MOD_SIZE);
    ntwk_send(SESSION_KEY, oaep_encoded);
```

Listing 3: AES Session and Packets

```
    #define MAX_PACKET_SIZE 2048
    aes_ctx ks_session, ks_auth;
    uint8_t encryption_buffer[MAX_PACKET_SIZE-1];
    uint8_t decryption_buffer[MAX_PACKET_SIZE-1-(2*AES_BLOCKSIZE)];
    uint8_t iv[AES_IV_SIZE];
    size_t padded_size, stripped_size;
    hashlib_AESLoadKey(aes_key, ks_session, AES_KEY_SIZE<<3);
    hashlib_AESLoadKey(&aes_key[AES_KEY_SIZE], ks_auth, AES_KEY_SIZE<<3);

    // to encrypt for each packet
    padded_size = hashlib_AESPadMessage(<message>, <msg_length>, &encryption_buffer[
AES_IV_SIZE], SCHM_DEFAULT);
    hashlib_RandomBytes(iv, AES_IV_SIZE);
    hashlib_AESAuthEncrypt(&encryption_buffer[AES_IV_SIZE], padded_size, encryption_buffer
, ks_session, ks_auth, iv);
    ntwk_send(ID, encryption_buffer);
    // AuthEncrypt() appends a message integrity tag
    // to not use authenticated encryption, use AESEncrypt() instead.

```

```
18      // to decrypt for each packet
19      packet = ntwk_get();
20      hashlib_AESAuthDecrypt(&packet[1], packet_len-1, decryption_buffer, ks_session,
    ks_auth);
21      stripped_size = hashlib_AESStripPadding(decryption_buffer, packet_len-1,
    decryption_bufferr, SCHM_DEFAULT);
```

4.3 Server-Side Session Handling

On the server side it is a bit harder to document since you could be using a number of libraries: OpenSSL, Python3 cryptography, cryptodome, pycryptodome, and many more. You will have to consult the associated documentation for the library in question to see how to do it properly. Nonetheless, I will place samples from TI-Trek's login system, which decrypts an AES-encrypted user login token, and later exchanges the secret with RSA.

Listing 4: TI-Trek AES decrypt

```
1       from Cryptodome.Cipher import AES
2       import hmac
3
4       ...
5       # log_in is a member function of the Client class.
6       # Data is the received packet
7       def log_in(self, data):
8         iv = bytes(data[1:17])
9         ct = bytes(data[17:])
10        cipher = AES.new(self.key, AES.MODE_CBC, iv=iv)
11        padded_key = cipher.decrypt(ct)
12        padding = padded_key[len(padded_key)-1]
13        key = padded_key[0:-padding]
14        for dir in <player_directory>
15          hashed_pw=hashlib.sha512(bytes(key)).hexdigest()
16          if hmac.compare_digest(hashed_pw, <saved_sha512_of_session_token>]):
17            #log user in
```

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()]