CEX++ 0.14

An Introduction to the CEX Cryptographic Library

John Underhill - owner@vtdev.com

February 22, 2017

Welcome

This document is intended as a brief, yet evolving summary of the work completed to date in the CEX++ cryptographic library. It is not intended as a lengthy description of the ciphers and protocols contained in the library, but rather as an introduction to the library's current capabilities and a listing of its contents.

Standard implementations of protocols are named and briefly described, (though there are links to the help page and relevant external papers and standards), but less obvious or non-standard implementations are described here in more detail.

The help file is linked throughout, and each help page contains a technical overview, implementation recommendations, class and function purpose and parameters, and links to external documentation.

This is a living document and a part of the CEX++ help library, and will be updated with each version reflecting the current state of the library's progress, as it evolves into a more complete and expansive suite of cryptographic tools and protocols.

There is just one update left before the library becomes 1.0, and the preliminary work on the Symmetric portion of CEX has been completed.

The current update has a lot of new functions and features, as the library prepares for the introduction of Asymmetric cryptography like Ring-LWE, McEliece and GMSS.

New in 0.14 are a suite of (pipelined and parallelized) AEAD block cipher modes; EAX, GCM and OCB, the introduction of the message authentication code generator GMAC, and the ability to control and manage SIMD and multi-threading capable functions using the global integration of the ParallelOptions class.

There have also been additions and changes to the API of symmetric ciphers and cipher modes, improvements to existing protocols like message digests, MACs and TRNGs, the addition of constant time functions and cache management, and expansion and improvements to the documentation.

Design: Function and Form

Common API Patterns

This library uses virtual methods contained in interface classes and implemented as interfaces by the various protocol implementations. For example, the ICipherMode contains a set of virtual public property-style accessors and function calls used to initialize and operate the class. These functions are named in often overlapping and intuitive ways, for example any class that requires initialization with a key, (MAC, KDF, Cipher, DRBG...), will have an **Initialize(***ISymmetricKey***)** function, used to load the key and initialize the internal state.

A protocol capable of multi-threading will contain a ParallelProfile() accessor function which returns/modifies a ParallelOptions class. This options class automatically pre-calculates the ideal input block-size for the algorithm based on the algorithms requirements and systems hardware capabilities, but also allows for custom manipulation of parallel-processing features, and input sizes that trigger parallel processing.

A class that requires input updates like a message digest or MAC function includes an **Update(***Input, Offset, Length***)** function, if the algorithm requires finalizing, (including AEAD modes), a **Finalize(***Output, Offset, Length***)** function is used.

The overlap of names applies to functions across protocol boundaries, so a user who knows how to implement a message digest, will find a MAC function easy to use, just as a stream cipher shares the **Transform(***Input, InOffset, Output, OutOffset, Length***)** API with a standard or AEAD block-cipher mode.

If you are processing data using multi-threading or SIMD instructions, setup is identical on block-cipher modes, stream-ciphers, DRBGs, PRNGs, and message digests. Essentially these patterns are meant to facilitate simple and obvious implementation procedures.

There is a high degree of automation at work behind the scenes, complex tasks like multi-threading and SIMD parallelism are handled automatically, allowing for simplified controls. The implementations also contain advanced API subsets through companion classes like ParallelOptions, that allow for more granular control by the advanced user.

It is true that using virtual interfaces is a bit slower than using templates, that there is a cost incurred by a vtable lookup, but this can be reduced to the point that it hardly registers on performance, particularly when appropriately sized data is processed by a hash or cipher (say 16 -32kb per update or transform call). Also, impact can be further mitigated by instantiating protocols that load or encapsulate a Pseudo Random Function, with a strictly typed instance rather than using the virtual parent, (i.e. use SHA256* dgt, rather than IDigest* dgt when declaring an instance).

There are a couple of strong advantages to using virtual members, a more flexible model, a more obvious and cohesive organization of classes and functions, simplified translation to other languages, and perhaps most importantly, I believe it improves upon readability and ease of deployment.

Another feature of the library uses namespaces and using statements to give complete access to a class and any dependant classes through a single declaration of the target protocol. For example, if you create an instance of a block cipher, the light-weight interface and enumeration dependencies associated with that classes public API are included in the namespace; the BlockCiphers enumeration member, CryptoSymmetricCipherException exception wrapper, the Digests enumeration, and the ISymmetricKey and SymmetricKeySize classes. Everything you need to operate that class is loaded automatically, with no need to include header files or namespace prepending of the child class required. Dependencies that require more substantial class forms are declared in the definition files (.cpp).

Documentation

Unfortunately, it is all too often that I (and likely you as well), open a header file of some interesting looking project only to find that there is not a single note, comment, or instruction about the API. Add to this, that the API is often phrased in some algorithm specific crypto-speak, which may adhere well to the protocol designers draft of pseudo-code, but to an implementer, one who is not an expert on the algorithm, or even cryptography in general, it presents a difficult challenge.

Unfortunately, this is the point where many developers spend only an hour or two browsing the help file, (if there even is one), and failing the discovery of an immediate and obvious explanation and code example; they begin searching the internet for working examples of code, this is colloquially known among developers as... the 'paste and prey' strategy.

Now, there are some crypto libraries with excellent online help files, for example; I believe Crypto++ is the gold-standard of help libraries in this space, the documentation is a small self-contained encyclopedia of cryptographic functions and algorithms, with clear examples, and sensible descriptions and recommendations.

This is the target I am reaching for while authoring this library. Clear and obvious examples of usage, configuration recommendations and limitations, links to relevant papers and specifications, and a concise and instructive overview of the implementation.

The CEX++ documentation can be organized into various hierarchies; namespace, class, module, and name. Each class provides information on all publicly visible functions, their usage and parameters, a code example, and a technical overview.

The documentation is updated regularly, (with each update and internal evaluation cycle), and is expanding quickly as features are added at an accelerated rate during this still 'early stage' in the libraries development. The goal though remains constant, and that is to facilitate the simplified and safe deployment of cryptography among application developers, and to provide a powerful cryptographic instrument to the C++ application developers domain.

Coding Style

The libraries style and syntax is coded in a vanilla way, leaning towards a more formal Windows programmers style, similar to Crypto++.

Class and local variables are Hungarian notation with a class variable using an 'm_' prefix, ex. *m_blockSize*. Global constants use the all uppercase C style (const size_t MAX_ROUNDS = 38), function local constants are all uppercase but usually only a single 6 letter word.

Class and function names are formal, and the only mildly unusual style quirk is that I use formal function parameter names as well, ex. void **LoadState(***Digests KdfEngineType***)**.

This style has evolved in a way that I believe improves readability, (and with increased readability, comes fewer implementation errors, and more maintainable code).

Some C++ programmers have an affinity for snake_case_style, and I have seen code where everything, even class names are written this way. C++ however is not C, and that style evolved from a need to prefix function names with a class or organizational tag. One of the nice things about C++ *is* namespaces, allowing a more ordered and logical separation of the application domain, and with that the ability to choose a more intelligible user-friendly syntax.

Because there is no formal, universal C++ style, (proposed ones like Google's, which well... leaves a lot to be desired), it's mostly a matter of preference. In my experience, the best strategy is to use a style which makes the logic as obvious as possible, particularly in long procedural functions, or complex sections of code. If a variable is a

global, a constant, a method parameter, or property, this should be instantly recognizable as you debug the code.

Another feature of this style is meaningful naming conventions. Local variable names should be short and concise, but enough to express their basic purpose. I see many codes that are expressed in rigid mathematical terms, coinciding to the protocol designers' expression of the function. There is a place for this, but in many cases the overall programmatic logic of the expression is obscured by expressing the function this way, (e.g. functions with parameters like: F(Zq, x, y, k, N, n, m), do not belong in public space).

One should remember that most people who take the time to study these implementations will not be doing so by comparing them side-by-side with the formal paper, instead they will be like minded developers, curious implementers, and occasionally professional security researchers, (and the ones that are comparing to the paper, are likely knowledgeable enough that the expression is obvious). So, naming conventions matter, and language that best conveys the intent and internal flow of the function is the most likely to reveal problems, and prevent them.

API Rules and Objectives

Here's a partial list summarizing some of the development goals and guidelines

- Use meaningful names; wherever possible classes and functions should be self-documenting
- Overlapping API names; public functions and their parameters should be self-describing and where a same-purpose function is required, it is re-used across protocol boundaries, ex. Update(Input, Offset, Length), or Initialize(SymmetricKey).
- Wrap ciphers and protocols in re-usable and flexible interfaces; this enforces a common base functionality, simplified access, pattern coherence, and run-time flexibility
- Integrate SIMD parallelism wherever possible; default to the highest system-supported version (AVX2 or SSE3), use simplified cross-platform intrinsic wrappers (UInt128, UInt256, and ULong256 classes), and always provide a sequentially processed fallback
- Auto-calculate recommended input block sizes for ciphers and protocols by evaluating system capabilities and algorithm performance requirements (ParallelOptions class)
- Provide strict recommended keying input sizes to every keyed protocol (LegalKeySizes)

- Provide simplified access to primitives through constructor enumeration parameters and instantiation helper functions, ex. CBC(BlockCiphers::RHX, or, BlockCipherFromName::GetInstance(BlockCiphers::Rijndael)
- All protocols that take a cryptographic primitive as a constructor parameter, can be initialized with either an instance pointer of that primitive, or the primitives' enumeration name (which auto instantiates the primitive internally, and manages it throughout the instance lifetime, including destruction)
- Each public facing class and function is fully documented, with complex or original work thoroughly commented in the definition file (documentation can be removed for release compilation)

Block Ciphers

CEX Help: IBlockCipher, AHX, RHX, SHX, THX

IBlockCipher

The Block Cipher Interface class

Rijndael -RHX/AHX

A Rijndael Cipher extended with an (optional) HKDF powered Key Schedule.

AHX is the (recommended) Rijndael AES-NI implementation, RHX is a fallback on systems without AES-NI capability.

The Rijndael specification: NIST FIPS 197.

Serpent -SHX

A Serpent cipher extended with an (optional) HKDF powered Key Schedule.

The Serpent specification: Serpent A Proposal for the Advanced Encryption Standard

Twofish -THX

A Twofish Cipher extended with an (optional) HKDF powered Key Schedule.

The Twofish specification: Twofish A 128-Bit Block Cipher

Standard vs Extended Ciphers

There are two variations of each of the block ciphers implemented in the library, the standard version, which complies to the original cipher design; Rijndael, Serpent, and Twofish, and their extended cousins RHX, SHX, THX, (and the recommended AES-NI version of Rijndael: AHX).

The HX stands for HKDF eXtended, because each cipher has an alternative key schedule that uses the HMAC based Key Derivation function HKDF

Expand, rather than the standard key expansion routine native to that cipher.

There are several advantages to this implementation, perhaps most importantly is that it allows a more flexible and powerful security model. The HX ciphers use a more cryptographically secure expansion function to generate the internal round-keys than the native mixing functions used by standard implementations of these ciphers. When HKDF is used as the key schedule function, longer round-key arrays become possible without a potential loss of security that may arise via modification of the native key expansion methods. HKDF is well vetted, and generally regarded as a strong KDF, whereas the comparatively weak Rijndael expansion function has been proven susceptible to several serious timing and differential attack vectors. Further, a constant-time hash function, (particularly one using SIMD instructions), leaks no information about the cipher, and should strongly mitigate many of the most serious attack vectors discovered to date.

The secure key schedule can safely produce longer internal key arrays, which enables an increased number of transformation rounds. More cycles through the rounds function, creates a more thoroughly diffused output, which should also strongly mitigate several attacks on the cipher, (I think Rijndael should be at 22 rounds currently, or twice the best-known attack).

The recommended digest for the extended ciphers is 256bit SHA-2, though any of the implemented message digests are compatible and safe to use with HKDF. The best input cipher key-size to use in this configuration is 64 bytes of key, and for added security, use a secret and random distribution code.

Some people think that a 512bit key is unnecessary, and in terms of brute force attacks that could be true, even when quantum computers halve the key space, 128 bits should be secure. That is assuming that attacks using quantum computers are never improved upon, and that no new attacks are discovered that further reduce the key-space. That doesn’t seem like a very wise or prudent position, given the escalating history of attacks thus far, and the ever-accelerating rate of technological advancement. If we want to maintain security margins for long-term security, then I believe larger keys will soon become necessary.

Another thing to consider, is that the extended ciphers rely on a hash function as the primary PRF, in this model, the best key size is the one that offers the most secure execution of the function, and this does not necessarily coincide with the brute-force minimum resistance required of the keys length.

Instantiation

The choice of cipher option, *standard* or *expanded*, is chosen at instantiation through the constructor, by passing either an instance of a message digest, or the digests enumeration name.

Using the digests enumeration name, the digest is instantiated internally using the

DigestFromName::GetInstance helper function. This instance is destroyed automatically when the class is finalized.

AHX(Digests KdfEngineType = Digests::None, size_t Rounds = 14);

You can also pass a digest instance (or null) to a constructor, this instance must be destroyed by the caller.

AHX(IDigest *KdfEngine, size_t Rounds = 14);

If the Digests enumeration name is set to 'None', the standard cipher configuration is used, likewise if the IDigest instance is initialized with 0.

If the Digest instance is loaded the cipher will be run in extended mode. The digest function is used to power the HKDF Expand function used by the key schedule during initialization.

The constructor is also where the number of transformation rounds are set, if the cipher is in extended mode, the number of rounds can be increased, in standard mode the cipher will process the standard defined number of rounds. The available rounds for a block cipher or stream cipher in extended mode are defined in the **LegalRounds()** function. The maximum legal rounds for Rijndael in AHX/RHX is 38 (AES256 is 14), and double the maximum standard rounds in Serpent at 64, and Twofish at 32 rounds.

Initialization

The **LegalKeySizes**() accessor contains an array of SymmetricKeySize structs, each containing an allowed KeySize, NonceSize, and InfoSize value. These sizes are auto-calculated based on the mode; standard mode generates the standard sizes (16, 24, 32, plus a 512bit key byte-size of 64 bytes), and the extended cipher calculates keys based on the message digests block size.

In extended operation mode, the 1st key size is always half the digests input block size, the 2nd key size is the digests block size (recommended), and the third is twice the block size, which triggers HKDF Extract during the key expansion phase.

The **DistributionCode()** array maps to the Info string used by HKDF Expand, this string is used to fill out the input block processed by the HMAC.

For additional security, it is recommended that this code be secret and random, and **DistributionCodeMax()** in length.

The optimal length for the info parameter is calculated by subtracting the HKDF 8bit internal counter plus the length of the digests finalizer padding (if any), from the ideal input block size used by HMAC. This guarantees that the digest finalizer processes one full block of data, including any finalizer code appended to the block. If, for example, you input one full block (128 bytes) to 512bit SHA-2 and call the finalizer, it will process the full block, and then process a second block of zero-byte padding and a 17-byte finalizer code. The reasoning for not processing padding is obvious; mixing routines in hash functions are primarily additive, that is they use multiplication, addition, and shifts or rotations to mix the state. Applying any of these functions by '0' does nothing to alter the state, so at best, you are wasting CPU cycles, and at worst... you could be leaking information.

The key and info parameters can also be added through a member of the ISymmetricKey virtual class. The *Key::Symmetric* namespace contains two key container classes; SymmetricKey and SymmetricSecureKey. Both classes contain keying material used by any cipher or protocol in the library that requires keyed initialization. The keying material is loaded in the key containers constructor, and accessible by the read-only accessors Key(), Nonce(), and Info().

The SymmetricSecureKey, encrypts the arrays when they are loaded through the constructor, and only decrypts them when read by the process through the accessor functions. The keying material is encrypted by collecting process-specific data from the system, (along with an optional Salt value), compressing this with 512bit SHA-2, and using the output to key a AES256 in CTR mode used to encrypt the keying material. If the key should somehow make it to disk, or it's memory region accessed by another process, all that can be recovered is values encrypted by an ephemeral and process-bound key.

SIMD Processing

The AHX, SHX, and THX block cipher implementations are all capable of processing data in parallel using SIMD instructions, (RHX is a lookup table based implementation of Rijndael, that is used a fallback for older systems that do not posses the AES-NI instructions).

The Twofish and Serpent implementations can use either 128bit or 256bit SIMD to process either 4 or 8 calculations in parallel. For example, the Serpent bit-slicing 'Sbox' functions can process four or eight integer operations in parallel as part of a completely SIMD parallelized rounds function.

This parallelism is achieved by using the SIMD wrapper classes UInt128, and UInt256.

These classes contain the various mathematical operators, load and store, and common utility functions implemented with either SSE3 or AVX-2 SIMD instructions. Common parallelizable functions are also implemented, like **XORBLK** in the IntUtils class.

AHX, the AES-NI implementation transforms data using the 128bit instructions native to that instruction set.

Each cipher implementation includes the **Tramsform64()** and **Transform128()** functions, called by a cipher mode to execute a transform using SIMD parallelism. Engaging parallel processing is largely an automatic operation; if the system supports multi-threading and/or SIMD, it is enabled by default, and activated by a cipher mode by processing a parallel-block length of data.

Block Cipher Modes

CEX Help: ICipherMode, CBC, CFB, CTR, ECB, ICM, OFB

The standard block-cipher modes all comply with the modes described in NIST SP800-38A, Recommendation for Block Cipher Modes of Operation.

2001 Edition Cipher Modes of Operation

ICipherMode

Block Cipher Standard Modes virtual interface class

CBC

An implementation of a Cipher Block Chaining Mode

CFB

An implementation of a Cipher FeedBack Mode

CTR

An implementation of a Big-Endian segmented Counter Mode

ECB

An implementation of an Electronic CodeBook Mode

ICM

An implementation of a Little-Endian Integer Counter Mode

OFB

An implementation of an Output FeedBack Mode

ParallelOptions

The **ParallelProfile()** property function provides access to a block cipher modes multi-threading and SIMD processing options through an instance of the ParallelOptions class. All the AEAD modes, as well as the ECB, CTR, ICM, and the CBC Decrypt function of the standard modes can be completely parallelized (SIMD and multi-threading), and the CFB Decrypt function can be multi-threaded.

The parallel options class auto-detects the ideal parallel block-size based on the systems capabilities (L1 Cache size, core count, and SIMD availability). The **ParallelBlockSize()** and **IsParallel()** read-only accessors are implemented on every parallel capable protocol in the library. The IsParallel property indicates if the algorithm and system are capable of multi-threaded processing, and the ParallelBlockSize is the input block-size that when passed to a Transform function, triggers parallel processing.

A **ParallelOptions** class is initialized by the constructor of the containing class, for example, a CBC instance initializes the options using the underlying ciphers input block-size, an SIMD capabilities flag, a working set offset size, and a processing channel-type flag.

ParallelOptions(size_t BlockSize, bool SimdMultiply, size_t ReservedCache, bool SplitChannel)

The SIMD Multiply flag tells the Calculate function that SIMD is available for that protocol, and multiply **ParallelMinimumSize()** by the highest available SIMD integer multiplier, (most of the symmetric functions use 32bit unsigned integers, so, if SSE3 is available, multiply the base block size by 4, if AVX2 is detected, multiply by 8).

The reserved cache size is the total byte size of a working set; hot tables, buffers, and class level variables used by the calling class. This is subtracted from the parallel block calculations, and tables are pre-cached by the ciphers to reduce cache evictions and the impact of timing variations.

The split channel parameter tells the Calculate function that this is either an algorithm that uses two equal size channels, like the input and output arrays processed by a cipher, or a single channel mechanism like a MAC or hash function. Split channel algorithms halve the final parallel block size to reduce pressure on L1 cache.

The number of physical and virtual CPU cores, the total L1 Data Cache size, as well as SIMD capabilities are collected using the CpuDetect class, which uses the cpuid function to poll the processor for these values. The base parallel-block size is initially the L1 data cache size, minus the reserved cache, and halved if the algorithm uses two channels. The resulting size subtracts the modulus of the parallel minimum-size to estimate the ideal parallel input block-size.

The number of processor cores assigned to multi-threading, the SIMD profile, and parallel input size can be set through the ParallelOptions class

properties, and the adjusted sizes are then calculated by calling the Calculate function.

The cipher modes use this information to schedule threads, calculate internal offsets and working set sizes, and channel processing through the selected SIMD functions… all done automatically.

Each time the cipher mode is Initialized, it checks for changes to the core properties; ParallelBlockSize, IsParallel, and **ParallelMaxDegree**. If a change is detected, the new sizes are evaluated and re-calculated.

Transform

The Transform function is the primary gate to the ciphers internal data transformation functions. The main function has a length parameter, the two secondary Transform functions have no length specified, so they expect an array of either BlockSize() or ParallelBlockSize() if IsParallel() is set to true, and return the number of bytes processed.

The library uses standard vectors exclusively, no C style arrays. This was done both to simplify the API (no array length parameters required), and for the benefits of using vectors; a rich API, auto-deleting, and safe.

Modes may also have secondary access to the data transformation using the non-virtual functions EncryptBlock and DecryptBlock. These functions process a single BlockSize segment in sequential mode, and are used primarily for internal functions.

The recommended Transform function:

```
void Transform(const std::vector<byte> &Input, const size_t InOffset, std::vector<byte> &Output, const size_t OutOffset, const size_t Length)
```

If the system is multi-core, and/or SIMD capable, and an input/output block of ParallelBlockSize or greater is passed to this function, multi-threading and SIMD parallelism is performed automatically.

AEAD Modes

CEX Help: IAeadMode, EAX, GCM, OCB

IAeadMode

An AEAD Cipher Mode virtual interface class

EAX

An Encrypt and Authenticate AEAD Block Cipher Mode as outlined in: The EAX Mode of Operation.

GCM

A Galois/Counter Authenticated Block Cipher Mode described in: The Galois/Counter Mode of Operation.

OCB

An Offset CodeBook Authenticated Block Cipher Mode as described in RFC 7253: The OCB Authenticated-Encryption Algorithm

Parallelization

The IAeadMode interface inherits the ICipherMode interface class used by the standard modes, so all the API is mirrored in these authenticated modes. This includes the Transform functions and parallel related accessors, so using an authenticated mode is nearly an identical operation.

Each of the authenticated modes (GCM, EAX, and OCB), are implemented with various levels of parallelism. The EAX and GCM modes use a parallelized and pipelined CTR mode to transform data. The OCB mode uses SIMD and a parallel loop to parallelize data processing.

The ideal parallel block-size is auto-calculated just as with the standard modes, and tuning of those modes is identical.

Finalize and Verify

The Finalize function generates the authentication tag, and writes it to an output array, this is true in either encryption or decryption. In encryption mode, the tag is usually appended to the output stream. In decryption mode, the tag is checked for equivalence against the tag received with the cipher-text.

The Verify function can be used in place of Finalize when decrypting a stream. It computes the Mac code internally, and compares it to the Input array parameter using a constant-time function.

AutoIncrement and PreserveAD

When enabled, AutoIncrement treats the Nonce loaded at Initialize as an incrementing monotonic counter. The initial nonce value received through the SymmetricKey container is stored, and each time the mode is finalized and the authentication code generated, the nonce is incremented and re-processed to create initial state. The output tag and cipher-text when using this feature is identical to an output generated by passing the values of an incremented nonce in cleartext and appending it to the cipher-text, just as it is recommended in RFC 5116: An Interface and Algorithms for Authenticated Encryption. The nonce is reset and the state re-initialized just as if the mode were re-initialized with an updated nonce.

The advantage to using the internal counter, rather then prepending it to each cipher-text, is that the updated nonce no longer has to be sent in the clear. A single starting nonce can be set along with the key when the mode is initialized. Though these modes claim to be safe even if the nonce is known to an adversary, allowing an adversary access to the nonce value must be considered a reduction in security, as the nonce value must be known to successfully decrypt and generate the correct authentication tag. An auto incrementing mechanism also creates simplified deployments, and reduces the risk of (potentially catastrophic) nonce re-use.

If auto incrementation is turned off, you can still re-initialize with a new nonce post-finalization, by using the Initialize function with a SymmetricKey input containing only an updated nonce array, this also bypasses the underlying block ciphers Initialize function.

Associated Data can be set after initialization, but before the first plain-text or cipher-text block is added, using the SetAssociatedData(Input, Offset, Length) function. If the PersistAD option is set, this data can be loaded when processing the first data segment, and will then be added automatically to each subsequent initialized segment of the stream.

Block Cipher Padding

CEX Help: IPadding, ISO7816, PKCS7, TBC, X923, ZeroPad

Padding modes are used to pad, and calculate offsets on partial input blocks when using standard block-cipher modes that require a full block of input, (CBC, CFB, and OFB).

IPadding

The Padding Modes virtual interface class

ISO7816

The ISO7816 Padding Scheme

PKCS7

The PKCS7 Padding Scheme

TBC

The Trailing Bit Compliment Padding Scheme

X923

The X.923 Padding Scheme

ZeroPad

The Zero Padding Scheme

Stream Ciphers

CEX Help: IStreamCipher, ChaCha20, Salsa20

IStreamCipher

Stream Cipher virtual interface class

ChaCha20

A parallelized ChaCha stream cipher as described in: ChaCha, a variant of Salsa20.

Salsa20

A parallelized Salsa stream cipher implementation as described in: Salsa20 design.

Parallelization

Just as with the streaming block-cipher modes CTR and ICM, the stream ciphers can be fully parallelized using either 128bit of 256bit SIMD instructions. The ciphers are also multi-threading capable, and can transform data at an incredible rate.

The ParallelProfile accessor returns/modifies a ParallelOptions class, and there are ParallelBlockSize and IsParallel read-only accessors.

Transform

The primary encryption processors are accessed in the same way as a standard cipher mode using the identical Transform functions.

There is also an Initialize function, a LegalKeySizes array, IsInitialized, Enumerator and Name, and a DistributionCode and DistributionCodeMax properties, just as in a standard block-cipher mode implementation. This use of overlapping API facilitates an easy transition between block and stream cipher operation.

Deterministic Random Bit Generators

CEX Help: IDrbg, CMG, DCG, HMG

The Deterministic Random Bit Generators (DRBGs), are keyed RNGs, used as either standalone protocols to generate pools of pseudo-random bytes, or as the random source for CSPRNGs and the KeyGenerator class.

Counter Mode Generator (CMG)

This is a standard Counter mode block-cipher based generator; it uses a block-cipher to encrypt an incrementing monotonic counter to create a key-stream (CTR), which is the generators output.

CMG can use any of the block-cipher implementations in the library, including the extended HX versions.

The CMG generator is instantiated by passing either a block-cipher instance or enumeration name into the constructor; using the enumeration name auto-initializes the cipher instance and destroys that instance when the CMG class instance is finalized, whereas using the cipher instance parameter type, requires that the instance is deleted by the caller.

An optional message digest and entropy provider can also be initialized through the constructor, these enable the predictive resistance feature.

This generator is capable of full parallelization using SIMD instructions and multi-threading.

Digest Counter Generator (DCG)

This is a Digest counter generator based on the outline in NIST SP800-90A, some additions to that design have been made, including backtracking and predictive resistance, and a non-zero counter initialization through the addition of a Nonce parameter added during initialization.

HMAC Based Generator (HMG)

The HMG generator is based on the HMAC generator outline contained in NIST SP800-90A R1 document. There are a few significant improvements to that design, including a strong KDF mechanism used to regenerate initial state for the predictive resistance feature, the (optional) use of a random nonce used as the counter, and the implementation of the Info parameter through the initialization options, which turns a standard HMAC counter generator into an HKDF generator, (with a 64bit counter).

Predictive and Back-tracking Resistance

Predictive and backtracking resistance prevent an attacker who has gained knowledge of generator state at some time from predicting future or previous outputs from the generator.

The (optional) resistance mechanism uses an entropy provider to add seed material to the generator, this new seed material is passed through a derivation function along with the current state, the output hash is used to reseed the generator.

The default interval at which this reseeding occurs can be set using the ReseedThreshold() property; once this number of bytes or greater has been generated, the seed is regenerated automatically. Predictive resistance is strongly recommended when producing large quantities of pseudo-random (10kb or greater).

Key Derivation Functions

CEX Help: IKdf, HKDF, KDF2, PBKDF2

HMAC Key Derivation Function (HKDF)

The HKDF Expand and Extract key derivation protocols as described in RFC 2104.

Key Derivation Function 2 (KDF2)

The Key Derivation Function version 2 as outlined in ISO18033-2.

Passphrase Based Key Derivation Function 2 (PBKDF2)

The Passphrase Based Key Derivation Function as described in RFC 2898.

SCRYPT

The Scrypt KDF, this will be implemented in 1.0.

Message Authentication Code Generators

CEX Help: IMac, CMAC, GMAC, HMAC

CMAC

The CBC MAC generator as described in RFC 4493.

GMAC

The Galois MAC generator as outlined in NIST SP800-39B.

HMAC

The Hash based MAC generator as specified in RFC 2104.

(Cryptographically Secure) Message Digests

CEX Help: IDigest, Blake256, Blake512, Keccak256, Keccak512, SHA256, SHA512, Skein256, Skein512

In release 1.0, each of the implemented hash functions will be capable of parallelized tree-hashing, and implemented with (optional) SIMD instructions.

BLAKE2

The 256 and 512bit variants of the Blake2 message digest as outlined in: Blake2 simpler, smaller, fast as MD5.

KECCAK

The SHA-3 winner Keccak as described in: The Keccak SHA-3 submission

SHA-2

512 and 256bit variants of SHA-2 as described in NIST Fips Pub 180-4: The Secure Hash Standard.

SKEIN

512 and 256bit variants of the Skein message digest, as described in: The Skein Hash Function Family.

True Random Number Generators

CEX Help: IProvider, CJP, CSP, ECP, RDP

The True Random Number Generators are a source provider of quality random bits, used to create keying material for ciphers and keyed cryptographic protocols.

CPU Jitter Provider (CJP)

The CPU jitter based entropy provider measures discreet timing differences in the nanosecond range of memory access requests and CPU execution time. Because the CPU and cache memory are continuously being accessed by various operating system and application processes, small timing differences can be observed and measured using a high-resolution timestamp.

Delays caused by events like external thread execution, branching, cache misses, and memory movement through the processor cache levels are measured, and these small differences are collected and concentrated to produce the providers output.

The CJP provider should not be used as the sole source of entropy for secret keys, but should be combined with other sources and concentrated to produce a key using the KeyGenerator class.

Crypto Service Provider (CSP)

An implementation of an entropy source provider using the system secure random generator.

On a windows system, the RNGCryptoServiceProvider CryptGenRandom() function is used to generate output. On Android, the arc4random function is used. All other systems (Linux, Unix), use dev/random.

Entropy Collection Provider (ECP)

The Entropy Collection Provider is a two-stage entropy provider; it first collects system sources of entropy, and then uses them to initialize a block cipher CTR generator.

The first stage collects numerous caches of low entropy states; high-resolution timers, process and thread ids, the system random provider, and statistics for various hardware devices and system operations.

These sources of entropy are compressed using Keccak to create a 512bit cipher key.

The key initializes an (HX extended) instance of Rijndael using 22 rounds and an HKDF(SHA256) key schedule.

The 16 byte counter and the HKDF distribution code (personalization string) are then created with the system entropy provider and the cipher initialized.

Output from the ECP provider is the product of encrypting the incrementing counter (key-stream).

RDRAND/RDSEED Provider (RDP)

The RdRand DRNG uses thermal noise to generate random bits that are buffered into a shift register, then fed into a CBC-MAC to condition the bytes.

The output from the CBC-MAC is obtained using the RDSEED API.

To accommodate large sampling, the system has a built in CTR_DRBG, (as specified in SP800-90), which is continuously reseeded with the output from RDSEED.

The output from the CTR DRBG is obtained using the RDRAND API.

There is some controversy surrounding the security of this mechanism, though the design appears to be sound, and has been reviewed by external auditors, it is still a proprietary closed system.

The entropy source itself must therefore be considered as a 'black box', a source that cannot be verified directly, and so must be considered to be of low entropy value.

For this reason, the DRNG should not be used as the sole source of entropy when creating secret keys, but should be used in concert with other sources of entropy.

Crypto Processors

CEX Help: FileStream, MemoryStream, CipherStream, DigestStream, MacStream

FileStream and MemoryStream

These classes wrap a file or byte array in a streaming interface in similar API as java and C# file and memory streaming interfaces.

CipherStream

The CipherStream class is an easy to use wrapper that initializes and operates a symmetric cipher, automating many complex tasks down to just a couple of methods, in an extensible ease of use pattern.

Either a block cipher and mode, or a stream cipher can be initialized through the classes constructor, using either the cipher (and options) enumeration members, or a cipher instance.

The CipherStream class uses the IByteStream interface, and can encrypt either a byte array using MemoryStream, or a file with FileStream.

This class supports parallel processing; if the cipher configuration supports parallelism (CTR/ICM, and CBC/CFB Decrypt), the IsParallel property will be set to true.

The IsParallel property can be overridden and set to false, disabling parallel processing.

If using the byte array Write method, the output array should be at least ParallelBlockSize in length to enable parallel processing.

DigestStream

Wraps Message Digest stream functions in an easy to use interface.

MacStream

Wraps Message Authentication Code (MAC) stream functions in an easy to use interface.

Tests and the Evaluation Project

The evaluation project contained in the /Win folder contains tests for every cipher and protocol in the library. Rijndael is tested with AESAVS certification vectors, Twofish using the official KAT vectors, and Serpent using the official Nessie vectors.

Standard cipher modes are tested using the KAT vectors from NIST SP800-38a, EAX, GCM and OCB, ChaCha and Salsa are all tested with official vectors published by the protocol designers.

All other protocols like Kdfs, message digests, and Mac functions are tested using the most rigorous official known answer tests available.

Parallel cipher-modes and message digests, along with every other advanced feature of the library are rigorously stress-tested for correct operation.

The example project also contains a cipher speed test, which tests both parallel and sequential modes of operation.

Project Roadmap

The Next Update: 1.0

Version 1.0 will be the first 'official' version of the library, in which the initial symmetric cryptography for the library will be considered complete. I think it is important to start with a strong foundation, and so this portion of the library, has, and will continue to evolve as the foundational elements in a larger and more comprehensive set of capabilities implemented in this project.

The first major version will include top-down rewrites of the message digest functions, including a flexible tree-hashing scheme that employs multi-threading and SIMD parallelism. These features will branch out through mechanisms like HMAC, which will be able to use the parallel forms of the digests.

A secure memory-resident vector class will be added and integrated into the various ciphers and protocols.

The Scrypt key derivation function will be added, and possibly a couple of other protocols as well, (I have written an AEAD mode based on some of the work done in Grøstl, that is much faster than GCM or OCB, and should be provably secure, but needs some finalizing and vetting by experts in that area, before I can consider integrating it into this library).

The Future

In post 1.0 versions I will be adding asymmetric ciphers, starting with Ring-LWE, and possibly McEliece, and eventually I hope to make this library a collection point for post-quantum resistant ciphers and signature schemes.

Once the core asymmetric ciphers are added, a network stack will be implemented, and a serious effort will be made towards interoperability with the major Linux distros, iOS, and android operating systems.

The next step will be the integration of a key exchange protocol that is designed specifically for peer-to-peer communications, and includes defences against various attack vectors, and uses a flexible and powerful authentication scheme, that requires minimal server involvement and is capable of ad-hoc deployment by the clients themselves.

Finally, I will add a TLS key exchange that uses post-quantum resistant ciphers and signature schemes.

.. and Beyond

I consider CEX as a vehicle towards what I see as the imminent necessary changes that must be made to the public key crypto-system. These changes

not only include the movement towards post-quantum secure cryptography, but fundamental changes to the design of the certificate authentication scheme, the public-key exchange mechanism, and in fact the entire paradigm must shift, as we transition through critical escalating changes to our technological capabilities.

Currently, there seems to be little if any accountability applied to certificate providers in this system, the most widely used asymmetric ciphers and signature schemes will soon be vulnerable to attacks from quantum computers, and I believe this framework must be redesigned in such a way that it provides a more flexible and powerful security model, and guarantees the continuous reliability of a long-term security infrastructure.

At some point, when changes to the library are primarily maintenance driven, and I am afforded the time to examine these problems in depth, I would like to get involved with designing a system like this, and explore the various orbiting frameworks like secure DNS and routing protocol security. But, like any other journey, it is taken one step at a time, and I can imagine no better way to explore these technologies, to understand them, than to write implementations of the protocols, and when possible, attempt to improve upon them.

As to why I chose cryptography as *my* journey; I strongly believe that maintaining a secure communications infrastructure is a fundamental requirement to establishing and preserving a free society. Without the ability to communicate freely, unfettered by state interference or interdiction, we risk losing the ability to organize against unlawful or unjust governance.

The ability of governments to collect and store the data generated by it's citizens is being executed in ever-increasing scope and detail, as technology matures and coalesces to create more powerful collection and analysis instruments. I see advances like quartz coins that can store hundreds of terabytes of data indefinitely, and I wonder where that technology will be in 20 years, and how the evolution of our technological capabilities will affect the future of our world.

Soon quantum computers will eclipse the supercomputers of today, and this will bring the ability to not just collect and store massive amounts of information, but for far more advanced and performant big-data analysis capabilities.

Autonomous AI 'bots' combing through massive amounts of data, creating associations, pattern matching, sorting and combining, increasingly more sophisticated and autonomous, and all of that data, the history of each of us, can be stored indefinitely.

Whatever the course technology leads us on over the next century, we are at a critical juncture right now, where the stable, long-term security of our communication systems is at risk by ever-accelerating technological discovery, and where the failure to secure those systems now, may have impact that spans many generations to come.

So… that's CEX++ 0.14

February 22, 2017

John G. Underhill