The Polymorphic Medley Cipher: 128 bit block length, 128 .. 1024 bit key length

C. B. Roellgen, PMC Ciphers, Inc.

20.10.2012

Abstract

Ever since the invention of the Polymorphic Cipher, the highly variable concept has caused a noticeable amount of fear in the publically financed security sector which reacted hastily and violently through commentators who tried to shrug the concept off. The underlying idea of - at minimum - selecting a cipher from a set of conceptually different ciphers in a keyed operation, is although as simple as it is effective and a similar and undisputed concept has even been implemented in very popular encryption products since at least 20 years.
So far the goal of PMC Ciphers was to create ultimate ciphers that could not be broken at all. For that reason, block sizes need to be excessively big and preferentially variable. So far a Polymorphic Cipher that is designed sufficiently close to keyed cipher selection and that utilizes widely used royalty-free cipher primitives like Anubis or the commonly known AES Rijndael encryption functions was missing.
In order to make a demonstration of a cipher of ciphers available, the royalty-free 128 bit Polymorphic Medley Cipher is from now on available for cryptanalysis and for use by everybody who wishes to implement the cipher in any product for civil use.

Key words: polymorphic, encryption, cipher, cascade, block, size, key, plaintext, ciphertext, cipher block chaining, CBC, electronic codebook, ECB, initialization vector, AES, Rijndael, Twofish, Serpent, Cast-256, RC6, SEED, Camellia, Anubis, hash, compression, SHA-256, Whirlpool, RIPEMD, Tiger, HAVAL-256, combined secrecy system, pseudorandom number generator, PRNG.

1. Introduction

In 1999 I've invented a cipher that was compiled from the user-supplied key and I called the idea "Polymorphic Cipher" as the different ciphers always came with the same interface, but with different code to perform the task to encrypt data. If a polymorphic cipher cannot be compiled - which today is prevented by many microprocessor platforms though DEP (Data Execution Prevention) in order to prevent viruses from doing malicious things, it is still possible to use totally variable round functions or to simply select a cipher from a set of base ciphers and to cascade a number of encryption operations. The latter concept is widely known to be a real useful feature in data encryption software.
As an example, a popular open-source Disk Encryption Software named TrueCrypt allows users to select the cipher from a set of three ciphers with a similar interface:
- AES Rijndael
- Serpent
- Twofish

It is further possible to select the following cascades:
- AES-Twofish
- AES-Twofish-Serpent
- Serpent-AES
- Serpent-Twofish-AES
- Twofish-Serpent

The fact that the user selects the cipher or the cascade of ciphers makes the selection operation a so-called "keyed operation". Due to high amount of entropy in ciphertexts produced by commonly used encryption algorithms like Serpent, an attacker cannot distinguish ciphers by analyzing large amounts of ciphertext. An attacker thus needs to try each cipher if he doesn't know the keyphrase.

Cascades of ciphers may consume a bit more CPU time, but an attacker can as well not distinguish between a single cipher or a cascade of ciphers. C.E. Shannon [1] provides the background for this.

TrueCrypt although only allows to choose from eight different ciphers (ciphers and cascades) and only two combinations of triple encryption are provided. According to [3] and [4], single and double encryption feature almost the same attack security.

Wouldn't it make sense to always cascade - let's say - eight ciphers from a set of (e.g.) eight ciphers like AES Rijndael, Serpent or Anubis?
Of course it this would make sense, simply because the math looks challenging for attackers and cryptanalysts!
Having to try 8 encryption functions is certainly a difficult task (TrueCrypt), but the need to make a guess between e.g. 40.320 for a cascade of 8 ciphers (every base cipher is guaranteed to be used once in the cascade) or even 16 million encryption functions (arbitrary selection of base ciphers) is a task that is more difficult by several orders of magnitude and even if half of the "cipher primitives" were considered as being "weak" or "broken", the cascade would still provide for a good safety margin. It is evident that the sequence in the cascade cannot be set in a dropdown menu any more. It makes much more sense to include this operation in the key setup function. Actually the cascade provides for at least 15 additional password bits (8!).

The key setup function is actually the decisive weakness of ciphers like AES Rijndael as this function executes very fast (850 clock cycles for 128 bit keys on a Pentium Pro microprocessor [2]). Twofish needs ten times longer - from the standpoint of an attacker a disaster.

What if key setup took hundreds of millions of clock cycles?
For a smart card chip application, a cipher with such characteristics would be useless, but a billion instructions are crunched by CPUs of modern smartphones or desktop PCs within a second or less.
The average user would probably feel a slight delay until a data connection was established, but an attacker would suddenly be deprived of the most common attack - the brute force attack using a dictionary.

It is logical that the performance of a 128 bit cipher of ciphers is limited by the comparably small and fixed block size. Cascade block ciphers can although very well increase attack security over any of the implemented base ciphers (AES Rijndael, Twofish, Serpent, Cast-256, RC6, SEED, Camellia and Anubis) [3] and [4].
Attack security is finally what it's all about.

2. The cipher

Recent work [3] proves that "for the wide class of block ciphers with smaller key space than message space, a reasonable increase in the length of the cascade improves the encryption security".
By using base ciphers with identical key space and message space, encryption security is likely to be very high.

The cipher is a cascade block cipher with eight 128 bit base ciphers all operated in 128 bit key length mode. The minimum key length is 128 bit in 8-bit words (16 bytes). Maximum key length is 1024 bit. Block size is exactly 128 bit in 8-bit words (16 bit). All base ciphers feature an identical interface through the use of wrapper functions. As an example, here's the wrapper function for the Anubis encryption function:

```
void CIPHER_PRIMITIVE_ENCRYPT_Anubis128(void  * pCC,uint8 * p128bit_Plaintext,uint8 * p128bit_Ciphertext)
{
	struct crypto_primitives::NESSIEstruct_anubis * pAnubis_cc;

	pAnubis_cc=(crypto_primitives::NESSIEstruct_anubis *)pCC;
	crypto_primitives::NESSIEencrypt(pAnubis_cc,p128bit_Plaintext,p128bit_Ciphertext);
}
```

The interface of the Polymorphic Medley Cipher consists of a key setup function, a basic encryption function, a corresponding decryption function and a function that frees the random access memory that holds the Internal State of the cipher. An alternative ECB mode encryption function as well as an encryption function for CBC mode is provided as well.

The key setup function

```
int PMCMED_keysetup(uint8 * pKey,void * pPMCMED_cc,uint32 key_length_in_bits,uint32 complexity)
```

initializes the crypto context (pointer `pPMCMED_cc` to the struct supplied as the second parameter with the key (pointer `pKey` supplied as first parameter). The key length as well as a complexity parameter are as well provided. key length is the number of key bits (must be a multiple of 8). The complexity parameter is provided to allow key setup to be considerably fast, but also very slow. Values in the range of 0 .. 256 make the function execute fast and values up to 65535 slow the function down. In the latter case, a multitude of keyed operations involving all base ciphers and hash functions are called many times in order to compute the Internal State of the Polymorphic Medley Cipher.

Three sets of encryption/decryption functions - two for data encryption in ECB (Electronic Code Book) mode and one data encryption in CBC (Cipher Block Chaining) mode exist:

The encryption function

```
void PMCMED_encrypt_cascade(void * pPMCMED_cc,uint8 * pPlaintext,uint8 * pCiphertext)
```

executes all base ciphers one after the other with different keys in an order that is set by the key setup function. The sequence of eight ciphers is set by the key setup function. Each base cipher is guaranteed to be used exactly one time in the cascade. The number of possible cipher combinations is exactly 40.320.
The decryption function

```
void PMCMED_decrypt_cascade(void * pPMCMED_cc,uint8 * pCiphertext,uint8 * pPlaintext)
```

executes the cascade in reverse order.

The encryption function

```
void PMCMED_encrypt_max_var_cascade(void * pPMCMED_cc,uint8 * pPlaintext,uint8 * pCiphertext)
```

executes eight base ciphers that operate with different keys one after the other in an order that is set by the key setup function. It is very well possible (with a probability of exactly 1/16777216) that the very same base cipher (e.g. AES Rijndael) is executed eight times in a row with different keys, but there is nothing wrong about that. There exist exactly 2^{24} = 16777216 different and equally probable cipher combinations.
The decryption function

```
void PMCMED_decrypt_max_var_cascade(void * pPMCMED_cc,uint8 * pCiphertext,uint8 * pPlaintext)
```

executes the cascade in reverse order.

Developers can either use the functions `PMCMED_encrypt_cascade()` / `PMCMED_decrypt_cascade()` OR

`PMCMED_encrypt_max_var_cascade()` / `PMCMED_decrypt_max_var_cascade()`. The advantage of the first set of encryption/decryption functions is that all base ciphers are executed one after the other. The disadvantage is the limited number of combinations for the cascade. The second set of encryption/decryption functions is likely to be advantageous due to optimum attack security as approximately 24 bit of variability are present rather than only 14 bit.

The encryption function

```
void PMCMED_CBC_encrypt_cascade(void * pPMCMED_cc,uint8 * pPlaintext,uint8 * pCiphertext)
```

performs CBC encryption of any number of consecutive blocks of data. It executes eight base ciphers that operate with different keys one after the other in an order that is set by the key setup function and that is modified for each block through the use of a scheduler encryption function. There exist exactly 2^{24} = 16777216 different and equally probable cipher combinations for each encrypted block.

The decryption function

```
void PMCMED_CBC_decrypt_cascade(void * pPMCMED_cc,uint8 * pCiphertext,uint8 * pPlaintext)
```

executes the base ciphers in reverse order.

In order to be able to use the CBC encryption functions properly, CBC mode MUST be initialized once and at any point of time when synchronization to a stream of data is required - e.g. once per video frame, by calling the function

```
void PMCMED_init_CBC_mode(void * pPMCMED_cc,word64 CBC_block_counter_start_value=0LL)
```

The unsigned 64 bit integer number `CBC_block_counter_start_value` can be initialized with any value that identifies a certain section of a data stream in order to further increase attack security.

The function

```
int PMCMED_free_memory(void * pPMCMED_cc)
```

must be called as soon as the cipher is not needed any more in an application software in order to deallocate the Internal State of the cipher.

2.1 Key Setup

During the key setup phase is the key expanded for all eight base ciphers multiple times. In addition to this, function pointers to the base ciphers and hash functions are initialized and permutated.

The following data is derived from the user-provided key:

- Sequence of function pointers to base hash functions
- Sequence of function pointers to base cipher functions
- 16 different Internal States for the base cipher functions
- Initialization Vector for Cipher Block Chaining (CBC) mode
- Selection of a base cipher that is used as scheduler and Initialization Vector for the scheduler

The 16 different Internal States for the eight base ciphers requires approximately 154 kBytes of RAM, which forces an attacker to provide this costly hardware multiple times in order to mount a distributed attack.

The key setup function uses the compression functions SHA-256, Whirlpool, RIPEMD, Tiger and HAVAL-256 to compute a pseudorandom sequence of these hash functions as well as a pseudorandom sequence of all eight base ciphers, then to compute hash results, to further swap function pointers to the base ciphers, to initialize the scheduler for CBC operations and finally to initialize a set of cipher contexts for the base ciphers - 16 for each base cipher.

2.2 Encryption/Decryption in ECB mode with cascades consisting of the entire set of base ciphers

For the encryption and decryption in ECB (Electronic Code Book) mode, one set of cipher contexts is selected at the end of the key setup function. The same function determines the sequence of ciphers that are later executed in a cascade by the EBC mode encryption and decryption functions `PMCMED_encrypt_cascade()` and `PMCMED_decrypt_cascade()`. Each base cipher is executed exactly once in the cascade at any position in the queue. Eight base ciphers are thus executed one after the other. The ciphertext of the first base cipher is the plaintext of the next base cipher in the queue and so on.

This is the source code of the encryption function:

```
void PMCMED_encrypt_cascade(void * pPMCMED_cc,uint8 * pPlaintext,uint8 * pCiphertext)
{
        int i,j;
        struct PMCMED_cipher_context    *       pPMCMED_cipher_ctx;

        pPMCMED_cipher_ctx=(struct PMCMED_cipher_context *)pPMCMED_cc;
        j=pPMCMED_cipher_ctx->ciphertext_scheduler[0] & (NUM_OF_CRYPTO_CONTEXTS_PER_CIPHER_DELEGATE-1);
        // let's select a certain set of contexts
        for (i=0;i<NUM_OF_CIPHER_FUNCTION_DELEGATES;i++)
        {
                pPMCMED_cipher_ctx->encryption_func_delegates[i](
                        (void *)&pPMCMED_cipher_ctx->pcc[((i*NUM_OF_CRYPTO_CONTEXTS_PER_CIPHER_DELEGATE)+j)
                        *MAX_SIZE_OF_CRYPTO_CONTEXT_IN_BYTES],pPlaintext,pCiphertext);
                i++;
                pPMCMED_cipher_ctx->encryption_func_delegates[i](
                        (void *)&pPMCMED_cipher_ctx->pcc[((i*NUM_OF_CRYPTO_CONTEXTS_PER_CIPHER_DELEGATE)+j)
                        *MAX_SIZE_OF_CRYPTO_CONTEXT_IN_BYTES],pCiphertext,pPlaintext);
        }
        memcpy(pCiphertext,pPlaintext,16);
        memset(pPlaintext,0xaa,16);
}
```

The function looks up the set of cipher contexts to use and subsequently encrypts the plaintext repeatedly with all available base ciphers. There exist n! cipher combinations (8 * 7 * 6 * 5 * 4 * 3 * 2 * 1 = 40.320). The advantage of executing all available base ciphers in an arbitrary sequence is that the entire set of base ciphers is definitely being used. There exist although only 40.320 possible combinations for cascades.

The decryption function executes the ciphers in reverse order.

2.3 Encryption/Decryption in ECB mode with cascades consisting of an arbitrary combination of base ciphers

For the encryption and decryption in ECB (Electronic Code Book) mode, one set of cipher contexts is selected at the end of the key setup function. The same function determines the sequence of ciphers that are later executed in a cascade by the EBC mode encryption and decryption functions `PMCMED_encrypt_cascade()` and `PMCMED_decrypt_cascade()`. Any base cipher can be selected for any position in the queue. Eight base ciphers are thus executed one after the other and the probability for a single base cipher being selected for all positions in the queue is 1/16777216 = 0.0000000596046. The ciphertext of the first base cipher is the plaintext of the next base cipher in the queue and so on.

This is the source code of the encryption function:

```
void PMCMED_encrypt_max_var_cascade(void * pPMCMED_cc,uint8 * pPlaintext,uint8 * pCiphertext)
{
        int                                     i,j;
        struct PMCMED_cipher_context    *       pPMCMED_cipher_ctx;
        int                                     index_arr[NUM_OF_CIPHER_FUNCTION_DELEGATES];

        pPMCMED_cipher_ctx=(struct PMCMED_cipher_context *)pPMCMED_cc;

        for (i=0;i<NUM_OF_CIPHER_FUNCTION_DELEGATES;i++) index_arr[i]=(
                pPMCMED_cipher_ctx->ciphertext_scheduler[i]>>4) & (NUM_OF_CIPHER_FUNCTION_DELEGATES-1);

        j=pPMCMED_cipher_ctx->ciphertext_scheduler[pPMCMED_cipher_ctx->ciphertext_scheduler[0] & 0x0f] &
                (NUM_OF_CRYPTO_CONTEXTS_PER_CIPHER_DELEGATE-1);
        for (i=0;i<NUM_OF_CIPHER_FUNCTION_DELEGATES;i++)
        {
                pPMCMED_cipher_ctx->encryption_func_delegates[index_arr[i]](
                        (void *)&pPMCMED_cipher_ctx->pcc[((index_arr[i]
                        *NUM_OF_CRYPTO_CONTEXTS_PER_CIPHER_DELEGATE)+j)
                        *MAX_SIZE_OF_CRYPTO_CONTEXT_IN_BYTES],pPlaintext,pCiphertext);
                i++;
```

```
                pPMCMED_cipher_ctx->encryption_func_delegates[index_arr[i]](
                        (void *)&pPMCMED_cipher_ctx->pcc[((index_arr[i]
                        *NUM_OF_CRYPTO_CONTEXTS_PER_CIPHER_DELEGATE)+j)
                        *MAX_SIZE_OF_CRYPTO_CONTEXT_IN_BYTES],pCiphertext,pPlaintext);
        }
        memcpy(pCiphertext,pPlaintext,16);
        memset(pPlaintext,0xaa,16)
}
```

The function looks up the set of cipher contexts that are to be used, initializes an array that contains indexes that point to certain base cipher functions and subsequently it encrypts the plaintext repeatedly with the previously selected base ciphers. There exist 2^{24} = 16777216 cipher combinations. The advantage of selecting base ciphers without any restriction is the large number of equally probably combinations for the cascade.

The decryption function executes the ciphers in reverse order.

2.4 Encryption/Decryption in CBC mode with cascades consisting of an arbitrary combination of base ciphers

In Cipher Block Chaining mode, blocks of data are encrypted/decrypted one after the other with each data block depending on the ciphertext of the previously encrypted block. It is thus possible to further add variability for the cipher during encryption/decryption.

CBC mode requires the initialization of a data buffer which holds the ciphertext generated by the previous encryption of a data block with an Initialization Vector IV as there is no previously generated ciphertext available in the first place. Additionally, a block counter can be initialized, e.g. with the frame number of an encrypted video stream or a timestamp in an audio file, etc. This mechanism allows to randomize encryption operations so that the encryption of static data, but with different values for the block counter, result in (ideally) indistinguishable ciphertext. The CBC encryption/decryption functions utilize this block counter value internally to alter the selection of base ciphers prior to each and every block encryption. The default value is 0 when CBC mode is initialized using this function:

```
void PMCMED_init_CBC_mode(void * pPMCMED_cc,word64 CBC_block_counter_start_value=0LL)
{
        struct PMCMED_cipher_context    *       pPMCMED_cipher_ctx;

        if (!pPMCMED_cc) return;
        pPMCMED_cipher_ctx=(struct PMCMED_cipher_context *)pPMCMED_cc;

        pPMCMED_cipher_ctx->CBC_block_counter=CBC_block_counter_start_value;
        memcpy(pPMCMED_cipher_ctx->last_block_CBC,pPMCMED_cipher_ctx->IV_for_CBC,16);
}
```

Encryption in CBC mode of any number of consecutive data blocks of data is performed through this encryption function:

```
void PMCMED_CBC_encrypt_cascade(void * pPMCMED_cc,uint8 * pPlaintext,uint8 * pCiphertext)
{
        struct PMCMED_cipher_context    *       pPMCMED_cipher_ctx;
        word64                                  w64buf;
        int                                     i,j;
        int                                     index_arr[NUM_OF_CIPHER_FUNCTION_DELEGATES];

        if (!pPMCMED_cc) return;
        pPMCMED_cipher_ctx=(struct PMCMED_cipher_context *)pPMCMED_cc;

        w64buf=pPMCMED_cipher_ctx->CBC_block_counter;
        pPMCMED_cipher_ctx->CBC_block_counter++;

        memcpy(pPMCMED_cipher_ctx->plaintext_scheduler,pPMCMED_cipher_ctx->initial_plaintext_scheduler,16);
        pPMCMED_cipher_ctx->plaintext_scheduler[7]^=(uint8)(w64buf & 0xff);
        pPMCMED_cipher_ctx->plaintext_scheduler[1]^=(uint8)((w64buf>>8) & 0xff);
        pPMCMED_cipher_ctx->plaintext_scheduler[5]^=(uint8)((w64buf>>16) & 0xff);
        pPMCMED_cipher_ctx->plaintext_scheduler[4]^=(uint8)((w64buf>>24) & 0xff);
        pPMCMED_cipher_ctx->plaintext_scheduler[3]^=(uint8)((w64buf>>32) & 0xff);
        pPMCMED_cipher_ctx->plaintext_scheduler[2]^=(uint8)((w64buf>>40) & 0xff);
```

```
        pPMCMED_cipher_ctx->plaintext_scheduler[6]^=(uint8)((w64buf>>48) & 0xff);
        pPMCMED_cipher_ctx->plaintext_scheduler[0]^=(uint8)((w64buf>>56) & 0xff);

        // let's now generate 128 bit that are impossible to guess. We'll derive from that data the sequence
        // of the ciphers
        pPMCMED_cipher_ctx->encryption_func_of_scheduler(
                pPMCMED_cipher_ctx->crypto_context_of_scheduler,
                pPMCMED_cipher_ctx->plaintext_scheduler,pPMCMED_cipher_ctx->ciphertext_scheduler);

        for (i=0;i<NUM_OF_CIPHER_FUNCTION_DELEGATES;i++) index_arr[i]=
                (pPMCMED_cipher_ctx->ciphertext_scheduler[i]>>4) & (NUM_OF_CIPHER_FUNCTION_DELEGATES-1);
        for (i=0;i<NUM_OF_CIPHER_FUNCTION_DELEGATES+(pPMCMED_cipher_ctx->ciphertext_scheduler[0]
                & 0x000f);i++)
        {
                j=index_arr[0];
                index_arr[0]=index_arr[(i+pPMCMED_cipher_ctx->ciphertext_scheduler[i & 0x0f])
                        & (NUM_OF_CIPHER_FUNCTION_DELEGATES-1)];
                index_arr[(i+pPMCMED_cipher_ctx->ciphertext_scheduler[i & 0x0f])
                        & (NUM_OF_CIPHER_FUNCTION_DELEGATES-1)]=j;
        }

        j=pPMCMED_cipher_ctx->ciphertext_scheduler[0] & (NUM_OF_CRYPTO_CONTEXTS_PER_CIPHER_DELEGATE-1);
        // perform CBC now
        for (i=0;i<16;i++) pPlaintext[i]^=pPMCMED_cipher_ctx->last_block_CBC[i];

        // encrypt with unknown sequence of ciphers
        for (i=0;i<NUM_OF_CIPHER_FUNCTION_DELEGATES;i++)
        {
                pPMCMED_cipher_ctx->encryption_func_delegates[index_arr[i]](
                        (void *)&pPMCMED_cipher_ctx->pcc[((index_arr[i]
                        *NUM_OF_CRYPTO_CONTEXTS_PER_CIPHER_DELEGATE)+j)
                        *MAX_SIZE_OF_CRYPTO_CONTEXT_IN_BYTES],pPlaintext,pCiphertext);
                i++;
                pPMCMED_cipher_ctx->encryption_func_delegates[index_arr[i]](
                        (void *)&pPMCMED_cipher_ctx->pcc[((index_arr[i]
                        *NUM_OF_CRYPTO_CONTEXTS_PER_CIPHER_DELEGATE)+j)
                        *MAX_SIZE_OF_CRYPTO_CONTEXT_IN_BYTES],pCiphertext,pPlaintext);
        }
        memcpy(pCiphertext,pPlaintext,16);
        memcpy(pPMCMED_cipher_ctx->last_block_CBC,pCiphertext,16);
        memset(pPlaintext,0xaa,16); // let's disguise our last intermediate result
}
```

The function first modifies a 128 bit pseudorandom number with the block counter and encrypts this number through one of the base ciphers that is used as a "scheduler". The resulting ciphertext is nothing but a stream of pseudorandom numbers that determine which of the eight base ciphers is executed at what time in the queue of eight cipher slots. Above of this, a set of cipher contexts is selected once per function call from the result of the "scheduler" encryption operation. This function allows for optimum attack security as almost any operation is influenced by a keyed operation.

The decryption function

```
        void PMCMED_CBC_decrypt_cascade(void * pPMCMED_cc,uint8 * pCiphertext,uint8 *
pPlaintext)
```

performs the same steps, but it executes the base ciphers in reverse order and performs the CBC operation (as a matter of logic) at the end.

3. Attack security and performance

The security of cascades has been an open question until 2006/2009. The security of cascades of $l \geq 3$ block ciphers improves significantly over single or double encryption ($l = 1$ or $l = 2$) [3].

Gazi and Maurer write in [3]: "In a recent paper [4], Bellare and Rogaway have claimed a lower bound on the security of triple encryption in the ideal cipher model. Their bound implies that for a block cipher with key length k and block length n, triple encryption is indistinguishable from a random permutation as long as the distinguisher is allowed to make not more than roughly $2^{k+1/2\min[n,k]}$ queries."

In our case k equals n, which yields for the advantage $2^{3/2*k}$, which is significant! Cascading only three ideal 128 bit block ciphers with 128 bit key length can be as secure as a 192 bit block cipher. AES Rijndael, Twofish, etc. are certainly not ideal ciphers, but they are certainly still a good choice to realize a cipher cascade.

Gazi and Maurer [3] continue with "This bound is significantly higher than the known upper bound on the security of single and double encryption, proving that triple encryption is the shortest cascade that provides a reasonable security improvement over single encryption. Since a longer cascade is at least as secure as a shorter one, their bound applies also to longer cascades. They formulate as an interesting open problem to determine whether the security improves with the length of the cascade also for lengths $l > 3$."

Due to the fact that the Polymorphic Medley Cipher always makes 8 calls to several ciphers out of a set of 128 bit encryption functions, the time that it takes to encrypt one block of 16 bytes (128 bit) is roughly 8 times longer than the average time it takes to encrypt a single block with AES Rijndael, Anubis, Twofish, Serpent, etc.

Attack security is tightly linked to speed - especially to the key setup time. This is typically the weak point of ciphers that are heavily promoted by government organizations whose mission is to spy on people.

Key setup for AES only "costs" several hundred instructions. A single core on a modern microprocessor can perform 2.89 million key setups per second!

The Polymorphic Medley Cipher is although designed for a long and adjustable key setup time. Key setup on a single core of a modern microprocessor can take between 6 .. 120 milliseconds, which allows for fast, as well as very secure operation. The longer the key setup time, the more computer power is required by an attacker to apply Brute Force or a Dictionary Attack or both.

Cipher	***Polymorphic Medley Cipher***	***Polymorphic Medley Cipher***	***AES (table-based)***	***AES (table-based)***
Type of machine code	32 bit C++ x86 code	64 bit C++ x64 code	32 bit C++ x86 code	64 bit C++ x64 code
Encryption speed on an Intel Core i7 950 clocked at 3.06GHz [Mbit/s]	119	135	605	1003
Minimum key setup rate on an Intel Core i7 950 clocked at 3.06GHz [key setups/s]	116	179	2.751.890	2.887.670
Maximum key setup rate on an Intel Core i7 950 clocked at 3.06GHz [key setups/s]	11	17	2.751.890	2.887.670
Encryption speed on an Intel Core 2 Duo T5750 CPU, clocked at 2.0GHz [Mbit/s]	81	n/a	394	n/a
Minimum key setup rate on an Intel Core 2 Duo T5750 CPU, clocked at 2.0GHz [key setups/s]	79	n/a	1.954.270	n/a
Maximum key setup rate on an Intel Core 2 Duo T5750 CPU, clocked at 2.0GHz [key setups/s]	7	n/a	1.954.270	n/a

Table 1: Encryption speed comparison: The Polymorphic Medley Cipher *vs. AES, desktop PC and laptop computer, compiler: Microsoft Visual C++ 2010*

4. Comparison of AES vs. Polymorphic Medley Cipher vs. The Polymorphic Giant Block Encryption Algorithm

Design goal	Polymorphic Giant Block Size Cipher	Polymorphic Medley Cipher	AES Rijndael
Large and variable block size	**Block size is only limited by the resources of the target computer(s). Target systems should run at 500MHz or higher and more than 10Mbyte free RAM should be available. The Strict Avalanche Criterion is thus met perfectly.**	**Not supported at all, but the approx. 10 times larger machine code and required RAM of 154kByte make the design more complex than AES alone.**	**Not supported at all. Ciphers like AES need little more than 1Kbyte of machine code and a microcontroller typically used in cheap smart cards and washing machines (approx. 20.000 transistors) to run. It is conceivable that such conventional ciphers could have been hardened against all kinds of attacks if more complex implementations would have been the target.**
No padding to reach block granularity shall be necessary	**Block size is totally variable and blocks keep their length => no padding required, which results in no information being transmitted in vein.**	**Like AES: 16 byte block granularity ⇨ Padding required**	**DES: 8 byte block granularity, AES: 16 byte block granularity ⇨ Padding required A 2048 bit conventional block cipher would require padding to 256 byte blocks resulting in dramatic increase in data traffic if used for the encryption of TCP or UDP data packets.**
Partitioning of extremely big blocks at arbitrary position	**Blocks that are too big to handle are truncated into sub-blocks with block sizes that are determined by the key as well as the length of the original block.**	**Not supported at all. Block size is fixed to 16 bytes just like AES.**	**Not supported at all. AES, DES and all other well-known block ciphers feature fixed block sizes.**
Resistance against all known attacks	**Due to its variable nature are Polymorphic Ciphers not susceptible to typical attacks that target specific characteristics and/or known weaknesses of fixed ciphers. Brute Force is although applicable to any cipher.**	**Design is more resistant than AES to Dictionary Attacks due to a long and irreducible key setup time (more than 100 million machine instructions). The cipher is bit more resistant against DPA (Differential Power Attack), but only because the complexity of the design.**	**AES can be broken easily by DPA (Differential Power Attack) on small microprocessors and micro-controllers [5].**
Resistance to future attacks that may cut effective key size by ½ or even 2/3	**Cutting of effective key size by ¾ would result in still extremely high complexity of O(2^{256}) or higher, which is regarded as totally safe for the next trillion years.**	**Cutting of effective key size by ¾ would result in still extremely high complexity of O(2^{256}), but only if long keys (1024 bit) are actually used.**	**Cutting of effective key size by ½ results in an extremely low complexity of 2^{64}. The cipher would be regarded as being broken. [6]**
Extremely long key setup time	**> 100ms on a modern microprocessor make comparably short keys safe against Brute Force attacks conducted on a few machines. Extremely long key setup time increases energy consumption multiplied by the time needed for Brute Force by factor 2.000.000.**	**2 .. 50ms on a modern micro-processor make medium-sized keys quite safe against Brute Force attacks if the attacks are conducted on a few machines.**	**<1µs help attackers to try each and every password combination. This is highly dangerous if short passwords are being used to protect data.**
Platform independence	**Runs on any 32 or 64 bit microprocessor or micro-controller.**	**Runs on any 32 or 64 bit microprocessor or micro-controller.**	**Runs on any 8-, 16-, 32- and 64 bit microprocessor and micro-controller.**
Polymorphism and data dependent selection of functions	**The cipher is not only completely variable, but also is the block size huge and unpredictable if truncation is performed. No static weakness is exhibited.**	**The cipher is variable, and there are no static weaknesses. The Cipher-Block-Chaining encryption function is even data dependent.**	**Classic ciphers are static and can thus be thoroughly reverse-engineered and analyzed. Cryptanalysis of a mechanism that does always exactly the same is somewhat easier than for a mechanism that never executes the same operation twice.**
Use of large amounts of resources	**1 Mbit internal state requires at least approx. 8 million transistor equivalents to run. This alone makes Brute Force Attack more difficult and much more expensive compared with conventional ciphers.**	**154 Mbyte of internal state need to be provided by an attacker. Mounting a Brute Force Attack on a large number of code breaker cores is much more expensive compared with conventional ciphers.**	Less than 50.000 transistor functions are required to build an AES block. **Approx. 1.000.000 AES blocks can run in parallel on an 8'' wafer to try and break a code using Brute Force.**
Attacks need to be	**The proposed cipher requires a lot of resources and extremely**	**The proposed cipher requires a lot of resources and extremely**	**Trying different AES keys requires 50.000 transistor equivalents and**

expensive for an attacker	much time for key setup, an attacker requires a "time x resources product" of approx. 200.000 times compared with AES Rijndael when using keys with a similar length.	much time for key setup, an attacker requires a "time x resources product" of approx. 100.000 times compared with AES Rijndael when using keys with a similar length.	less than 1µs. This isn't really all that much. This is a REAL weakness.
High speed	1500 Mbit/s on an Intel Core i7 950 (3.06GHz) (64 bit C++ code, 1024 byte block length)	135 Mbit/s on an Intel Core i7 950 (3.06GHz) (64 bit C++ code)	1000 Mbit/s on an Intel Core Core i7 950 (3.06GHz) (64 bit C++ code)
Proven security	Three round Luby Rackoff features proven security. Polymorphic encryption is increasingly popular among experts but it's probably impossible to prove security of the entire cipher.	Due to a relatively large number of conceptually different base ciphers like Anubis or Serpent or AES, known weaknesses of these base ciphers play no role. Cascades actually improve attack security noticeably [3] and [4]. This alone is sufficient to assume a higher attack security than for AES alone.	Security is not proven. Extensive peer review indicates that the cipher could be broken in the future: For 128-bit Rijndael, the problem of recovering the secret key from one single plaintext can be written as a system of 8000 quadratic equations with 1600 binary unknowns. [8] Recently has a new related-key boomerang attack on the full AES-192 and the full AES-256 been found by . Biryukov and Khovratovich [7]. A 256 bit key is reduced to a 119bit key when using AES-256. The attack is not applicable to 128 bit keys.
Licensing	Cipher is NOT open source and a license needs to be bought from PMC Ciphers, Inc.	Cipher is open source and royalty-free.	Cipher is open source and royalty-free.

Table 2: Comparison of key features of different ciphers

5. Conclusion

The proposed Polymorphic Medley Cipher is probably the first implementation of a cascaded cipher based on eight conceptually different and widely discussed base ciphers in order to increase attack security over single or double encryption. The base ciphers as well as the sequence of their execution is determined during key setup or even at runtime of the CBC encryption/decryption functions. The cipher is a Polymorphic Encryption Algorithm that gives an attacker no chance to know which base cipher has actually been used in an encryption operation and where in the queue. Attackers are deprived of constants and exhaustive sieve (Brute Force Attack) is impeded by a key setup procedure that consumes a lot of time. Parallelization of exhaustive sieve is hampered through the sheer amount of space on a silicon wafer required to implement the cipher.

As the cipher is royalty-free, open source, based on well-analyzed base ciphers and hash functions and as it's easy to use, it certainly makes sense to implement it in commercial software.

References:

[1] C.E. Shannon. Communication theory of secrecy systems. Bell System Technical Journal, 1949

[2] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, N. Ferguson. Performance Comparison of the AES Submissions. (1999)

[3] P. Gazi, U. Maurer. Cascade Encryption Revisited. ASIACRYPT 2009, LNCS 5912, pp. 37–51. (2009)

[4] M. Bellare, P. Rogaway. Code-Based Game-Playing Proofs and the Security of Triple Encryption. Eurocrypt 2006. LNCS, vol. 4004, pp. 409–426. Springer, Heidelberg (2006), http://eprint.iacr.org/2004/331

[5] S. Chari, C. Jutla, J.R. Rao, P. Rohatgi. A cautionary Note Regarding Evaluation of AES Candidates on Smart-Cards. http://citeseer.nj.nec.com/chari99cautionary.html, 1999

[6] Orr Dunkelman, Nathan Keller. A New Criterion for Nonlinearity of Block Ciphers. http://vipe.technion.ac.il/~orrd/crypt/apnp.pdf, 2006

[7] A. Biryukov, D. Khovratovich, Related-key Cryptanalysis of the Full AES-192 and AES-256, https://cryptolux.org/mediawiki/uploads/1/1a/Aes-192-256.pdf, 2009

[8] Nicolas T. Courtois, Josef Pieprzyk. Cryptanalysis of Block Ciphers with Overdefined Systems of Equations http://eprint.iacr.org/2002/044.pdf, 2002

For more information: http://www.pmc-ciphers.com