An Obfuscated Implementation of RC4

Roger Zahno and Amr M. Youssef

Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Quebec, Canada
r_zahno@ciise.concordia.ca, youssef@ciise.concordia.ca

Abstract. Because of its simplicity, ease of implementation, and speed, RC4 is one of the most widely used software oriented stream ciphers. It is used in several popular protocols such as SSL and it has also been integrated into many applications and software such as Microsoft Windows, Lotus Notes, Oracle Secure SQL and Skype.

In this paper, we present an obfuscated implementation for RC4. In addition to investigating different practical obfuscation techniques that are suitable for the cipher structure, we also perform a comparison between the performance of these different techniques. Our implementation provides a high degree of robustness against attacks from execution environments where the adversary has access to the software implementation such as in digital right management applications.

1 Introduction

Because of its simplicity, ease of implementation and robustness, RC4 [1] has become one of the most commonly used stream ciphers. In its software form, implementations of RC4 appear in many protocols such as SSL, TLS, WEP and WPA. Furthermore, it has been integrated into many applications and software including Windows, Lotus Notes, Oracle Secure SQL, Apple AOCE, and Skype. Although the core of this two decade old cipher is just a few lines of code, the study of its strengths and weaknesses as well as its different software and hardware implementation options is still of a great interest to the security and research communities.

Cryptographic techniques are traditionally implemented to protect data and keys against attacks where the adversary may observe various inputs to and outputs from the system, but has no access to the internal details of the execution. On the other hand, several recently developed applications require a higher degree of robustness against attacks from the execution environment where the adversary has closer access to the software implementation of key instantiated primitives. Digital Rights Management (DRM) is an example of such applications where one of the main design objective is to control access to digital media content. This can be achieved through white-box implementations where encryption keys are hidden, using obfuscation techniques, within the implementation of the cipher. White-box implementations for block ciphers, such as AES and DES, are widely available [2] [3]. However, to the authors' knowledge, there is no published white-box implementation for any dedicated stream ciphers including RC4. Directly applying the techniques developed for white-box implementation of block ciphers to stream ciphers does not seem to work, mainly, because normal operation of stream ciphers requires us to always maintain the inner state of the cipher. Recovering the inner state of the stream cipher usually sacrifices the security of the cipher even if the attacker is not able to recover the key. In fact, it seems to be an open research problem to determine if a practical white box implementation (in a cryptographic sense) would ever exist for stream ciphers such as RC4.

On the other hand, obfuscation can be used to transform a program from an easily readable format to one that is harder to read, trace, understand and modify. This offers an additional layer of security as it protects the program by increasing the required human and computational power that is needed to reverse engineer, alter, or compromise the obfuscated program. Obfuscation techniques can be categorized into automated and manual methods. The former systematically, using specific tools, modify the source code (e.g. [4]) or the binary executable file (e.g. [5],[6]). The latter techniques rely on the programmer to follow obfuscation techniques during coding. These techniques are governed by the programming language in use. For example, languages like C and C++, which are commonly used in the implementation of cryptographic primitives due their high performance, are flexible in syntax and allow the use of pointers.

Obfuscated programs can be reverse engineered using techniques such as static and dynamic analysis. Static analysis techniques analyze the program file by performing control flow and data flow analysis ([7],[8],[9]) without running the program. Dynamic analysis, on the other hand, takes place at runtime and addresses the followed execution path.

In this paper, we investigate several obfuscation techniques that are suitable for applications to array-based stream ciphers such as RC4. We also perform a comparison between the performance of these different techniques when applied to RC4. Although our proposed implementation does not provide the same level of theoretical security provided by white-box implementations for block ciphers, it still provides a high degree of robustness against attacks from execution environments where the adversary has access to the software implementation such as in digital right management applications.

The rest of this paper is organized as follows. A brief review of the RC4 key scheduling algorithm and pseudorandom generation algorithms is provided in the next section where we also briefly describe the Skype attempt to obfuscate the RC4 software implementation. Our proposed obfuscated implementation is described in section 3. Section 4 shows a performance comparison between different implementation options. Finally, our conclusion is provided in section 5.

2 The RC4 Cipher

In this section, we briefly review the Key Scheduling Algorithm (KSA), and the Pseudorandom Generation Algorithm (PRGA) of RC4. We also describe the Skype attempt to provide an obfuscated software implementation for RC4.

2.1 Standard RC4 Implementation

While traditional feedback shift register based stream ciphers are efficient in hardware, they are less so in software since they require several operations at the bit level. The design of RC4 avoids the use of bitwise operations as it requires only byte manipulations which makes it very efficient in software. In particular, RC4 uses 256 bytes of memory for the state array, $S[0]$ through $S[255]$, k bytes of memory for the key, $key[0]$ through $key[k-1]$, and two index pointers: a sequential index i, and quasi random index j.

Algorithm 1 shows the KSA of RC4, where the permutation S is initialized with a key of variable length, typically between 40 to 256 bits. Once this is completed, the key stream is generated using the PRGA shown in Algorithm 2. The generated key stream is combined with the plaintext, usually, through an XOR operation.

Algorithm 1. RC4 Key Scheduling Algorithm (KSA) [1]

```
1: for $i = 0 \to 255$ do
2:     S[i] := i
3: end for
4:
5: j := 0
6: for $i = 0 \to 255$ do
7:     j := (j + S[i] + key[i mod keylength]) mod 256
8:     swap values of S[i] and S[j]
9: end for
```

Algorithm 2. RC4 Pseudo-Random Generation Algorithm (PRGA) [1]

```
1: i := 0
2: j := 0
3: while GeneratingOutput do
4:     i := (i + 1) mod 256
5:     j := (j + S[i]) mod 256
6:     swap values of S[i] and S[j]
7:     K := S[(S[i] + S[j]) mod 256]
8:     output K
9: end while
```

Analyzing the KSA and PRGA algorithms of RC4 yields the following observations:

1. As with most stream ciphers, an adversary does not have to find the key in order to break the cipher. In other words, recovering the initialized inner-state S allows the adversary to efficiently generate the keystream output of the cipher and decrypt the target ciphertext even without knowing the *key* array.
2. An adversary who is able to observe the values of the index pointer j in the PRGA can efficiently recover the whole inner state S.

2.2 Skype's RC4 Implementation

The only reference to obfuscated RC4 implementation in the open literature appeared in a Blackhat publication that describes the leaked implementation used in Skype [10] [11]. By analyzing this leaked implementation, we observe the following:

- Regarding the cipher itself, the cryptographic key used is 80 bytes in length whereas standard implementations use a key of 40 to 256 bits in length (i.e. 5 to 32 bytes).
- Regarding key management, the Skype's implementation selects a key from a pool of 2^{32} keys.
- Regarding the use of the cipher, RC4 itself is used as an obfuscation technique to hide the network layer.

The implementation utilizes a macro called *RC4_round* that is used in both the KSA and the PRGA. Therefore, we first describe this macro. This macro is shown in Algorithm 3. When called, the macro is passed the following parameters:

i**:** the sequential index
j**:** the quasi random index
$RC4$**:** an array corresponding to S in the standard implementation
t**:** a variable for swapping the i^{th} and j^{th} element in S
k**:** the cryptographic key used in this iteration

Lines 1, 3 and 4 describe the swapping operation, line 2 evaluates the new quasi random index j, and line 5 evaluates the key for this iteration. The main difference in the use of this macro between KSA and PRGA is in the key value passed and the action based on the return value. In the PRGA, the value for k is always zero, whereas in the KSA, the key used in this iteration is passed. Furthermore, in the KSA, the returned value of this macro is discarded, whereas in the PRGA the returned value is used as part of the key stream.

Algorithm 3. Round Macro: RC4_round(i,j,t,k,RC4) [10]

```
1: t :=RC4[i],
2: j := (j + t + k) mod 256,
3: RC4[i] := RC4[j],
4: RC4[j] := t,
5: RC4[(RC4[i] + t) mod 256] {Output to be returned}
```

The KSA is shown in Algorithm 4. In this implementation, the inner-state of the cipher is stored in a data structure called rc4. This structure contains an array (representing the array S from Algorithms 1 and 2) which holds the random ordered values, a sequential index i, and quasi random index j. The "for" loop in lines 1 through 6 initializes the array rc4.s sequentially from 0 to 255. The array rc4.s is allocated exactly 256 sequential bytes in memory. The implementation takes advantage of the array structure in memory by initializing 4 bytes in each iteration rather than a single byte. Therefore, the index j in the loop is incremented in each iteration by 4 bytes ($0x04040404$), and is assigned to the array at the i^{th} position. Consequently, the index i has to be incremented by 4 in each iteration. The reordering step (lines 9 through 11) is done by the macro *RC4_round*.

Algorithm 4. Skype Implementation for the RC4 KSA [10]

```
 1: j := 0x03020100
 2: for $i = 0 \rightarrow 255$ do
 3:     i := i + 4
 4:     j := j + 0x04040404
 5:     rc4.s[i] := j
 6: end for
 7:
 8: j := 0
 9: for $i = 0 \rightarrow 255$ do
10:     RC4_round(i, j, t, byte(key,i%80), rc4.s)
11: end for
```

The PRGA is shown in Algorithm 5 where the ”for” loop iterates over the data stream (buffer, of size bytes) performing the encryption/decryption operation. This is done by XORing the data (in buffer) and the key stream generated by the macro *RC4_round*. Furthermore, the indices i and j in the data structure rc4 are updated.

Algorithm 5. Skype Implementation for the RC4 PRGA [10]

```
1: for $(;bytes;bytes--)$ do
2:     i := (i + 1) mod 256
3:     buffer++ {positioning the pointer to the new value to be en-/decrypted}
4:     *buffer = *buffer XOR RC4_round(i, j, t, 0, s)
5:     rc4.i := i
6:     rc4.j := j
7: end for
```

Clearly, the Skype implementation described above does not provide enough level of protection for the inner state of the cipher. It should be noted, however, that Skype uses the RC4 cipher itself as an obfuscation technique for the network layer but the effective data stream (voice, chat, video) is encrypted with AES [11].

3 Proposed Implementation

In this section we present our obfuscated implementation of RC4. Throughout our work, we assume that the cipher is implemented as a standalone module, i.e., the implemented code contains only the functionality of the cipher. Another approach would be to mix the implementation of the cipher with parts of a larger application. Such a needle in the haystack approach can add more security as the containing application offers more obfuscation space. However, since an implementation of this approach is highly dependent on the containing application, it is therefore not considered in this work.

In our proposed obfuscated implementation, we first eliminate the use of the array S by using independent set of variables. To improve the efficiency of this approach, we use function pointers. Following that, we utilize multithreading to provide security against dynamic analysis attacks. Finally, We present other generic techniques used to further obfuscate the proposed implementation. Throughout the remaining of this paper, for illustrative purposes, we use a toy implementation of RC4 (with an array of size $N = 4$). However, performance measures have been made based on a RC4 with standard parameters (i.e. with an array of size $N = 256$).

3.1 Eliminating the S Array Data Structure

As the KSA and PRGA algorithms show, standard RC4 implementation requires only a few data objects, namely two index pointers i and j, and an array S.

As a first step towards obfuscation, we substitute the array data structure S by N independent variables, where N is the number of elements in the array S. Unlike the elements of an array which are stored in consecutive memory locations, these independent variables can be scattered throughout the program memory. On the other hand, working with such independent variables eliminates our ability to dynamically address them using a loop structure since we no longer have an array index that can be related to loop counters. To address these variables, we use the loop unrolling technique, also known as loop unwinding. This is illustrated for the toy implementation in Figure 1. As depicted in the figure, the implementation is based on two nested switch/case structures, where the outer structure operates over the index i and the inner structure operates over the index j. It is worth noting that the inner switch/case structures are almost identical for various outer switch/case structures. However, since the i^{th} element in array S has been substituted with an independent variable, this variable has to be correctly referenced in each inner switch/case structure. This nested structure cannot dynamically evaluate expressions such as $S[S[i]+S[j]]$. For such expressions, a dedicated switch/case structure, $Switch(Output-Index)$, is used. This structure is passed an intermediate value, $Output-Index = S[i]+S[j]$, and evaluates the expression above. In an 8 bit implementation with an array of size $N = 256$, the number of possible combinations is given by $N \times N = 2^8 \times 2^8 = 2^{16}$. This implies that a total of 2^{16} case statements are needed to represent all the possible combinations. Thus, despite its conceptual simplicity, the use of nested switch/case structures results in a prohibitively large program (e.g. for $N = 256$, the program size exceeds 12 MB). In the next subsection we show how this obfuscation approach can be enhanced to yield a more practical program size.

3.2 The Use of Function Pointers

Function pointers are pointers that hold addresses of functions, and can be used to execute them. Depending on the address assigned to the pointer, a single function pointer

```
while(NOT_END_OF_DATA_STREAM)
{
    i++;
    Switch(i) {
        Case 0: //i = 0
            Switch(j) {
                j = (j + S0); //j = (j + S[i]) mod N;
                Case 0:    //j = 0
                           //swap(S[i], S[j]);
                           swap(S0, S0);
                           //Output S[S[i] + S[j] mod N]; → Output S[Output-Index];
                           Output-Index = (S0 + S0);

                Case 1: swap(S0, S1); Output-Index = (S0 + S1);
                Case 2: swap(S0, S2); Output-Index = (S0 + S2);
                Case 3: swap(S0, S3); Output-Index = (S0 + S3);
            } //end switch(j)

        Case 1: //i = 1
            Switch(j) {
                j = (j + S1); //j = (j + S[i]) mod N;
                Case 0: swap(S1, S0); Output-Index = (S1 + S0);
                Case 1: swap(S1, S1); Output-Index = (S1 + S1);
                Case 2: swap(S1, S2); Output-Index = (S1 + S2);
                Case 3: swap(S1, S3); Output-Index = (S1 + S3);
            } //end switch(j)

        Case 2: //i = 2
            Switch(j) {
                j = (j + S2); //j = (j + S[i]) mod N;
                Case 0: swap(S2, S0); Output-Index = (S2 + S0);
                Case 1: swap(S2, S1); Output-Index = (S2 + S1);
                Case 2: swap(S2, S2); Output-Index = (S2 + S2);
                Case 3: swap(S2, S3); Output-Index = (S2 + S3);
            } // end switch(j)

        Case 3: //i = 3
            Switch(j) {
                j = (j + S3); //j = (j + S[i]) mod N;
                Case 0: swap(S3, S0); Output-Index = (S3 + S0);
                Case 1: swap(S3, S1); Output-Index = (S3 + S1);
                Case 2: swap(S3, S2); Output-Index = (S3 + S2);
                Case 3: swap(S3, S3); Output-Index = (S3 + S3);
            } // end switch(j)

    } //end switch(i)

    Switch(Output-Index) {
        Case 0:    Output S0;//→ Output S[S[i] + S[j] mod N];
        Case 1:    Output S1;
        Case 2:    Output S2;
        Case 3:    Output S3;
    } //end switch(Output-Index)

} //end while
```

Fig. 1. The implementation of the PRGA when replacing the array data structure by independent variables

can be used to call multiple functions. Normally, a designated array is used to hold the addresses of the functions and when a function is to be called, its address is assigned to the pointer and the function is executed. A visualization of function pointers is shown

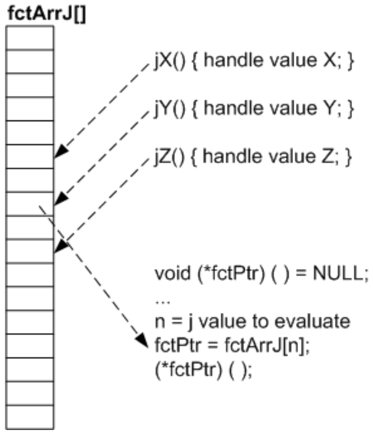

Fig. 2. Array of Function Pointers

in Figure 2 where the array $fctArrJ[]$ is the designated array that holds the addresses of functions $jX(), jY(), jZ()$. As the code fragment shows, the address of the desired function in loaded into the function pointer $fctPtr$, and then executed.

T. Ogiso *et al.* [12] analyze the use of function pointers in software obfuscation. Specifically, they prove that when using arrays of function pointers, determining the address a pointer points to is NP-hard.

Arrays of function pointers can be used to replace the inner switch/case structures in our obfuscation technique presented in the previous subsection. In addition to the security advantage resulting from the difficulty of determining the address a function pointer points to, implementing function pointers requires much less space than switch/case structures described in the previous section. This enables us to maintain the complexity introduced by the nested switch/case structure while reducing the program size.

The use of function pointers as a replacement of the inner switch/case statement requires the following:

1. For each index j, there exists a function in which the variable that substitutes $S[j]$ is hard coded. Furthermore, the variable that substitutes $S[i]$ is passed as a parameter to this function. With these variables, the functionality of RC4 can be easily realized.
2. There exists a designated array that holds the addresses of the functions described above.

The array of function pointers can be directly used to replace the inner switch/case structures operating on index j. The inner switch/case statements are replaced by functions in which the variable representing $S[j]$ is hard coded. To evaluate the inner structure, we first compute the new j value. This is used to retrieve the corresponding function address from the designated array. Finally, the retrieved function is called using a function pointer and the variable that substitutes $S[i]$ is passed as a parameter.

Figure 3 shows the use of function pointers as a replacement of the inner switch/case structures of our obfuscated toy implementation of RC4. As shown in the figure, the

```
//Declaration of function pointers
unsigned int (*output)() = NULL; //for functions returning the value of S[x]
void (*jN)(unsigned int) = NULL; //for functions handling a specific value of index j

//function returning the value of S[x]
oS0() { return S0; }
oS1() { return S1; }
oS2() { return S2; }
oS3() { return S3; }

// functions handling a specific value of index j
// The variable representing S[j] is know in advance and hardcoded
// The variable that handles S[i] has to be passed as function parameter to sX
J0(sX) { swap(S0,sX); output = arrayOutput[S0+sX]; Output output(); }
J1(sX) { swap(S1,sX); output = arrayOutput[S1+sX]; Output output(); }
J2(sX) { swap(S2,sX); output = arrayOutput[S2+sX]; Output output(); }
J3(sX) { swap(S3,sX); output = arrayOutput[S3+sX]; Output output(); }

// Assign addresses of the functions to the arrays
// Access to the functions via its position in the array
arrayJN[N] = {&J0, &J1, &J2, &J3};
arrayOutput[N] = {&oS0, &oS1, &oS2, &oS3};

while(NOT_END_OF_DATA_STREAM)
{

    i++;

    Switch(i){

        Case 0:   //i = 0
                  j = (j + S0); //j = (j + S[i]) mod N;
                  jN = arrayJN[j]; //Select function address from array
                  jN(S0); //Execute function via the function pointer

        Case 1:   //i = 1
                  j = (j + S1); //j = (j + S[i]) mod N;
                  jN = arrayJN[j]; //Select function address from array
                  jN(S1); //Execute function via the function pointer

        Case 2:   //i = 2
                  j = (j + S2); //j = (j + S[i]) mod N;
                  jN = arrayJN[j]; //Select function address from array
                  jN(S2); //Execute function via the function pointer

        Case 3:   //i = 3
                  j = (j + S3); //j = (j + S[i]) mod N;
                  jN = arrayJN[j]; //Select function address from array
                  jN(S3); //Execute function via the function pointer

    } // end switch(i)

} //end while
```

Fig. 3. Implementation of the PRGA using switch/case for *i* and array of function pointer for *j*

loop over index *i* remains unrolled using the switch/case structure proposed initially. The setup for the array of function pointers requires:

1. The declaration of two function pointers ($*output$ and $*jN$)
2. The definition of output functions $oS0(), oS1(), os2()$ and $oS3()$ that return the value $S[x]$

3. The definition of functions $j0(sX), j1(sX), j2(sX)$ and $j3(sX)$ that replace the inner switch/case structure
4. The definition of arrays used to hold the addresses of the functions stated in steps 2 and 3 above

Next, we illustrate the combination of switch/case structures with array function pointers. The outer structure used to unroll the loop over the index i remains unchanged. However, the inner switch/case structure is replaced by using the function pointers concept. To do this, we first compute the new j value. This is used to retrieve the corresponding function address from the designated array. Finally, the retrieved function is called using a function pointer where the variable that represents $S[i]$ is passed as a parameter. With these variables, the functionality of RC4 can be realized. When implemented, this approach while improving the obfuscation level, significantly reduces the obfuscated program size . In comparison to the initial attempt (in section 3.1), the program size is reduced from 12 MB to about 450 KB.

The techniques used so far have mainly increased the resilience of the implementation against static analysis. We, next, introduce further obfuscation with the objective of increasing its resilience against dynamic analysis.

3.3 Multithreading

Traditionally, multithreading allows various parts of a program to run simultaneously. Each such part is called a thread and although these are functionally independent, they share some resources such as processing power and memory, with other threads. Shared resources, such as data, code, and heap segments allow communication and functional synchronization between threads. Furthermore, constructs such as *critical sections*, and *semaphores* enable the realization of atomic units which, in turn, prevent corruption of shared resources. Because of this parallelism and the randomness in order of execution, analyzing multithreaded programs is much harder than their single threaded counterparts [13]. In this section, we capitalize on this and utilize the randomness in the order of execution introduced by multithreading to further obfuscate our implementation. To do so, we require the following:

1. There exists a multithreaded environment where each thread implements the RC4 functionality (key stream value) for a specific subset of index values i. That is, each thread contains an implementation of a subset of switch/case statement for the corresponding values of i. Furthermore, for each implemented switch/case statement, let the thread implement the function pointer concept for all values of j, as described in 3.2.
2. The sets of index values i are assigned to the threads such that each value of i is assigned randomly to at least two threads. This introduces randomness in the threads that have the capability of implementing the RC4 functionality for a given value of i. Since at least two threads have this capability, the execution path cannot be determined with certainty which introduces an additional layer of obfuscation.

If one uses only 2 threads, requirement 2 above would result in identical functionality for both threads, which simplifies conducting static analysis on the cipher. Thus, in our

```
Thread1() {

    while(NOT_END_OF_DATA_STREAM)

    {
        EnterCriticalSection; //enter the coherent, not interruptible code sequence

        Switch(i) {

            //Case 0: i = 0 is not handled in this thread

            Case 1:   //i = 1
                      j = (j + S1); //j = (j + S[i]) mod N;
                      jN = arrayJN[j]; //Select function address from array
                      jN(S1); //Execute function via the function pointer
                      i++;
                      //Leave the coherent, not interruptible code sequence
                      LeaveCriticalSection;

            Case 2:   //i = 2
                      j = (j + S2); //j = (j + S[i]) mod N;
                      jN = arrayJN[j]; //Select function address from array
                      jN(S2); //Execute function via the function pointer
                      i++;
                      //Leave the coherent, not interruptible code sequence
                      LeaveCriticalSection;

            Case 3:   //i = 3
                      j = (j + S3); //j = (j + S[i]) mod N;
                      jN = arrayJN[j]; //Select function address from array
                      jN(S3); //Execute function via the function pointer
                      i++;
                      //leave the coherent, not interruptible code sequence
                      LeaveCriticalSection;

            default:  //Handles the non-available 'cases'
                      //Leave the critical section
                      LeaveCriticalSection;

        } // end switch(i)
    } //end while
} //end Thread1

Thread2() {
    //same structure like Thread1(), but
    //Case 0: i = 0 is handled
    //Case 2: i = 2 is handled
    //Case 3: i = 3 is handled
    //and
    //Case 1: i = 1 is not handled
}

Thread3() {
    //same structure like Thread1(), but
    //Case 0: i = 0 is handled
    //Case 1: i = 1 is handled
    //Case 3: i = 3 is handled
    //and
    //Case 2: i = 2 is not handled
}
```

Fig. 4. Implementing the PRGA using multithreading

implementation, the minimum number of threads used is set to 3. This ensures that the implementations of various threads differ. The specific number of threads to be used is left as a design parameter.

To compute the key stream value for the current value of *i*, the running thread enters a critical section and retrieves *i*. If this thread does not implement the switch/case statement for this value of *i*, the critical section exits without affecting the cipher inner state and without producing any new keystream words. On the other hand, if the thread implements the switch/case statement for this value of *i*, the key stream value is evaluated and returned.

Figure 4 illustrates a toy implementation, with $N = 4$, of the multithreading implementation described above. In this example, the sequential index *i* can have the values $\{0,1,2,3\}$, and 3 threads are used. The first thread implements the switch/case statements for $i \in \{1,2,3\}$; the second thread implements the statements for $i \in \{0,2,3\}$, and the third thread implements the statements for $i \in \{0,1,3\}$. For standard RC4 parameters, this implementation increases the program size to 650 KB but offers an additional obfuscation layer and enhances the implementation's resilience to dynamic analysis attacks.

3.4 Handling the Key Scheduling Process

As shown in section 2.1, RC4 runs two main algorithms, the PRGA, and the KSA. While the implementation of the KSA can be obfuscated using the same techniques discussed above, one weakness of this approach is that the cryptographic key has to be passed in the clear to the KSA algorithm. In this section, we discuss a possible extension of the above implementation in order to mitigate this vulnerability.

In the white-box implementations of AES [2] and DES [3], the cryptographic key is integrated into the lookup tables of the implemented algorithms. Furthermore, the lookup tables are pre-created outside the users' environment. Applying this off-line generation technique to the inner states of RC4 can be used to eliminate the need for a KSA algorithm and consequently mitigate the vulnerability described above. This shifts our objective from protecting the key and key scheduling algorithm to protecting the process of securely assigning the off-line generated values to the inner-state.

Assume a setup where the user receives some encrypted data stream, and pre-created inner-state for the cipher from some service provider. In this case, the inner-state can be transferred from the provider to the customer in the form of an array. Instead of generating the inner-state by the KSA, the inner-state is initialized by directly assigning the values from the array to the corresponding variables. In other words, the array from the provider contains 256 values corresponding to *S* and the two index pointers *i* and *j* in a random order. Those 258 values are directly assigned to the variables representing the inner-state on the customer side. It should be noted that as long as the order of the values in the array is not known, an adversary cannot gain any useful information about the inner state of the cipher. Furthermore, assuming that the service provider knows the memory structure of the user's cipher, the service provider can produce a formatted memory dump that can be loaded directly into the user's cipher.

To this point, our obfuscation approaches structurally altered the implementation of the cipher. Additional obfuscation techniques, that are deployable on a smaller scale can also be utilized. These techniques do not significantly change the implemented structure and are easily applied. In the next subsection, we briefly explore the application of such techniques to our RC4 implementation.

3.5 Generic Obfuscation Techniques

In this section, we introduce a set of standard techniques that can be used to further obfuscate our implementation of RC4. These techniques are independent of the structure of the implemented program and do not significantly change the structure of the resulting obfuscated program.

3.5.1 Order and Dimension Change of Arrays

When using arrays of function pointers, assigning the pointers to the array in a sequential order introduces a $1:1$ mapping between the j value and the index of the array. This mapping can be further obfuscated using a random allocation table or by modifying the array structure. In [14], Zhu, *et al.* address this problem by changing the index order and the arrays' dimension. Transforming an array $A[N]$ into an array $B[M]$, where$M > N$ and M is relatively prime to N, can be done by applying the mapping $B[i] = A[i \times N \bmod M]$. We have used this methodology to obfuscate the array of function pointers in our implementation.

3.5.2 Variable Aliases

When two or more variables address the same memory location, they are called aliases. Introducing aliases to a program reduces the effectiveness of static analysis techniques as they increase the data flow complexity [7] since the attacker has to identify and track all aliases that manipulate a specific memory location. The larger the program, the harder it is for the attacker to identify and keep track of all the aliases. The pointers used in our implementation are an extensive form of aliasing, and therefore, introduce an additional level of obfuscation.

3.5.3 Scattering the Code for the Swap Operations

The RC4 cipher makes extensive use of swapping in both the KSA and the PRGA algorithms. In a standard implementation, monitoring the swapping function easily reveals the position of j and consequently, its value, which compromises the security of the implementation. To address this problem, in our implementation, we scatter the steps of the swap functionality throughout the program. In addition, we use a pool of temporarily swap variables rather than a single variable.

3.5.4 Opaque Constructs

Opaque predicates are expressions that evaluate either to true or false upon a given condition, but their outcome is known/controlled in advance. These constructs introduce confusion and are widely used in obfuscation [7], [15]. Opaque predicates can be classified based on their possible outcome into two types. In the first type, the outcome is always either 'true' or always 'false'. An example of such predicate would be $j > 255$, which is always evaluated 'false' in an 8 bit environment. In the second type, the output could be either 'true' or 'false', but is controlled by adjusting the statement that it evaluates. To increase the complexity of control flow analysis, we implemented a similar approach where either the real function or some other dummy is executed.

3.5.5 Evaluation of the Index Pointer j

Normally, the index pointer j, at step i, is calculated as

$$j_i = j_{i-1} + S[i] \tag{1}$$

where j_i denotes the value of the index pointer j at iteration i and $S[i]$ denotes the value of the inner state array during the i^{th} iteration of the cipher. Monitoring the value of j_i, while knowing the value of i, allows reproducing the inner state. In our implementation, we obfuscate the computation of j_i by introducing three intermediate variables (a, b, c) that are initialized as follows:

$$\begin{aligned} b &= \text{random}, \\ a &= b - j_{i-1}, \\ c &= 2a - b - S[i+1]. \end{aligned}$$

Then we calculate the value of j_i as $j_i = a - c$.

4 Performance Evaluation

In this section we compare the execution costs (i.e., program size and execution time) for various combinations of obfuscation techniques proposed in the previous sections.

The reported timing, as summarized in Table 1, are measured on an HP PC with a quad core Intel 2.67 GHz processor, and 8 GB of RAM. The prototype was implemented in C using Microsoft Visual C++ that was running on Windows 7 Enterprise platform. As expected, obfuscated implementations impose a penalty on the resulting program size and execution speed.

From Table 1, the slowdown factor varies highly between the obfuscation configurations. For configuration 1, the slowdown factor is about 16, whereas the slowdown factor when implementing all the described obfuscation techniques is about 481. The slowdown factors for the white-box implementations of AES and DES found in the published literature are 55 for AES and 10 for DES [16],[17].

In the following subsections, we highlight the main causes of this performance impairment.

Table 1. Program size and throughout for different obfuscation options

RC4 Implementation options	Program Size (KB)	Throughput (KB/sec)	a	b	c	d
No obfuscation	8	288,700	-	-	-	-
Configuration 1	514	17,850	x	-	-	-
Configuration 2	518	15,450	x	x	-	-
Configuration 3	537	11,500	x	x	x	-
Fully obfuscated	969	600	x	x	x	x

a: Loop unrolling over index pointer *i*, Array of function pointers, Variable aliases, Opaque constructs
b: Order and dimension change of arrays
c: Evaluation of index pointer *j*
d: Multithreading

4.1 Multithreading

Although multithreading is typically used to enhance the performance of a program, in our implementation it is used only as an obfuscation technique. Multithreading, as implemented, significantly enhances the programs resilience against dynamic analysis attacks but it also slows down the resulting program because of the following reasons:

1. The threads are essentially executed in sequence as the key stream value is evaluated only within a critical section. Therefore, when this section is in use by a given thread, other threads remain idle.
2. The use of a critical section introduces an additional overhead. Furthermore, as the design of RC4 dictates, only a single index can be evaluated at a time, which offers no room for parallelism.
3. Threads that do not implement the switch/case statement for the given value of the index *i* introduce an additional delay. These threads lock the critical section and consequently prevent other threads from operation and, yet, produce no keystream words.

4.2 Excessive Use of Context Switches

The proposed implementation extensively uses branch statements to switch between real and dummy functionality. This context switching introduces a delay since each switch/case structure is evaluated before the real functionality is executed. In addition, each function, whether dummy or real, introduces further delay as its parameters have to be pushed or pulled from the stack. Furthermore, due to the use of arrays of function pointers, in each iteration, two additional functions have to be handled.

4.3 Additional Calculations Overhead

Many mathematical calculations are required to obfuscate the value of *j* and the array indices. This includes the additional calculations used to obfuscate the index pointer *j*.

Furthermore, when obfuscating the order of arrays by changing their dimensions, additional computations are used to select the correct index for the address of each function.

5 Conclusion

In this work, different practical obfuscation techniques for RC4 implementation were investigated and compared. Our implementation provides a high degree of resiliency against attacks from the execution environment where the adversary has access to the software implementation such as digital right management applications. Furthermore, while the focus of this paper was RC4, these techniques can be applied to other array-based stream ciphers such as HC-128, HC-256 and GGHN.

Providing a white box implementation with more theoretical foundations for different cryptographic primitives, including RC4, is an interesting research problem. Exploring how to use multithreading to speed up the RC4 implementation is another challenging research problem.

References

1. Menezes, A., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press (1996)
2. Chow, S., Eisen, P.A., Johnson, H., van Oorschot, P.C.: White-Box Cryptography and an AES Implementation. In: Nyberg, K., Heys, H.M. (eds.) SAC 2002. LNCS, vol. 2595, pp. 250–270. Springer, Heidelberg (2003)
3. Chow, S., Eisen, P., Johnson, H., van Oorschot, P.C.: A White-Box DES Implementation for DRM Applications. In: Feigenbaum, J. (ed.) DRM 2002. LNCS, vol. 2696, pp. 1–15. Springer, Heidelberg (2003)
4. STUNNIX. C++ Obfuscator - Obfuscate C and C++ Code, `http://www.stunnix.com/prod/cxxo/overview.shtml` (accessed September 2011)
5. UPX: the Ultimate Packer for EXecutables, `http://upx.sourceforge.net/` (accessed September 2011)
6. SecuriTeam. SecuriTeam - Shiva, ELF Encryption Tool, `http://www.securiteam.com/tools/5XP041FA0U.html` (accessed September 2011)
7. Collberg, C.S., Nagra, J.: Surreptitious Software: Obfuscation, Watermarking and Tamperproofing for Software Protection. Addison-Wesley (2010)
8. Bergeron, J., Debbabi, M., Desharnais, J., Erhioui, M., Lavoie, Y., Tawbi, N.: Static detection of malicious code in executable programs. Int. J. of Req. Eng. (2001)
9. Wang, C., Hill, J., Knight, J., Davidson, J.: Software Tamper Resistance: Obstructing Static Analysis of Programs. Technical Report CS-2000-12. Univ. of Virginia (2000)
10. Reddit: the Front Page of the Internet. Skype's Obfuscated RC4 Algorithm Was Leaked, so Its Discoverers Open Code for Review: Technology, `http://www.reddit.com/r/technology/comments/cn4gn/skypes_obfuscated_rc4_algorithm_was_leaked_so_its/` (accessed September 2011)
11. Biondi, P., Desclau, F.: Silver Needle in the Skype, `http://www.secdev.org/conf/skype_BHEU06.pdf` (accessed September 2011)

12. Ogiso, T., Sakabe, Y., Soshi, M., Miyaji, A.: Software obfuscation on a theoretical basis and its implementation. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences E86-A, 176–186 (2003)
13. Collberg, C.S., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science. University of Auckland (1997)
14. Zhu, W., Thomborson, C.D., Wang, F.-Y.: Obfuscate arrays by homomorphic functions. In: GrC, pp. 770–773 (2006)
15. Collberg, C.S., Thomborson, C.D., Low, D.: Manufacturing Cheap, Resilient and Stealthy Opaque Constructs. In: POPL, pp. 184–196 (1998)
16. Park, J.-Y., Yi, O., Choi, J.-S.: Methods for practical whitebox cryptography. In: 2010 International Conference on Information and Communication Technology Convergence (ICTC), pp. 474–479 (November 2010)
17. Link, H.E., Neumann, W.D.: Clarifying obfuscation: Improving the security of white-box encoding, cryptology eprint archive. In: Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC 2005), vol. I (2005)