

A CPLD-based RC-4 Cracking System

Paul D. Kundarewich and Steven J.E. Wilton
Dept. of Electrical and Computer Engineering
University of British Columbia
Vancouver, BC, Canada
kundarew@ieee.org, stevew@ece.ubc.ca

Alan J. Hu
Dept. of Computer Science
University of British Columbia
Vancouver, BC, Canada
ajh@cs.ubc.ca

Abstract

This paper presents a CPLD-based system for cracking the RC4 encryption algorithm. The system achieves outstanding price/performance, easily beating other low-cost approaches such as commodity PCs. The system was implemented using a single Altera EPF10K20 Complex Programmable Logic Device (CPLD) (currently approx. CAD\$90) on an Altera UP1 Education Board. This CPLD is large enough to contain the control unit and five functional units. Measured performance on our prototype shows that we can crack 32-bit RC4 in 15 hours expected time (30 hours worst case). This gives a theoretical expected time of 159 days to crack 40-bit keys - the maximum possible key length that can be exported from Canada and the United States. Our result demonstrates the effectiveness of programmable logic (CPLD or FPGA) against even a cryptosystem designed for software implementation.

1 Introduction

RC4 is a proprietary stream cipher developed by Ron Rivest for RSA Data Security, Inc. The scheme is very widely used; among other things, it is commonly used to encrypt Web traffic under the SSL protocol. Although the encryption scheme is proprietary, details and source code were anonymously released to the public in 1994. This source code interoperates correctly with legally licensed versions of RC4, so it is widely believed to be a correct description of the RC4 algorithm. Our system cracks this published algorithm [1].

Encryption of a message using RC4 is accomplished by the generation of a pseudo-random number sequence which is XORed with the message to produce a data stream that looks random to outside observers. The message can be decrypted by simply XORing the encrypted data with the same pseudo-random number sequence. The sender and receiver generate this pseudo-random sequence by performing simple logical operations. The exact sequence of these operations depends on a *key*, a

short (typically 32 or 40 bits) sequence of bits. As long as both sender and receiver use the same key, they will generate the same pseudo-random sequence, and thus will be able to decrypt each other's messages. The secrecy of the message relies on the fact that no other observer knows the key being used by the sender and receiver, and thus can not decrypt the message.

One method to crack the encryption is to use a "brute-force" attack. In this kind of attack, all possible keys are generated and, for each key, the message is decrypted. The first few bytes of the resulting decrypted message is compared with a few bytes of a known correct message, and if there is a match, the correct key has been found, and the rest of the message can be decrypted. This method requires knowledge of at least the first few bytes of the unencrypted message, making it a *known plaintext attack*. In practice, known plaintext attacks are facilitated by the fact that many encrypted data transmissions start with known or easily deduced header information.

The difficulty with a brute force attack is that it scales exponentially with the key length. For example, with a 32-bit key, 2^{32} or 4.3×10^9 keys need to be tested while with a 40-bit key, 2^{40} or 1.1×10^{12} keys need to be tested. We estimate that a 200MHz Pentium Pro would take over 500 days to test all the keys within a 40-bit implementation.

Despite the fact that RC4 is well suited to a software implementation, a hardware implementation is likely to uncover the correct key much faster. A hardware implementation allows us to tailor the architecture specifically to the problem and allows us to implement several functional units, each of which can operate in parallel. This is especially appropriate for a brute force attack, since each functional unit can operate on a subset of the key space, independent of all other functional units.

This paper describes the design and implementation of such a circuit on an Altera CPLD. The resulting architecture is trivially scalable, and results in a significantly better price/performance tradeoff than a software implementation.

The paper is organized as follows. Section 2 describes the RC4 algorithm, Section 3 discusses the design and implementation of the circuit, and Section 4 details the results.

2 Algorithm

The heart of the RC4 algorithm is the generation of the pseudo-random sequence using the key. Figure 1 shows this portion of the algorithm. There are two 256-byte arrays, S and K . The S -array is initialized with $S_0 = 0, S_1 = 1, S_2 = 2, \text{etc.}$ The K -array is then initialized using the key. If the key is N bytes long, the first N bytes in K are copied directly from the key. This pattern is then replicated to fill the entire K -array. Once the two arrays have been initialized, the S -array is “randomized” by repetitively swapping entries, as shown in Figure 1.

Using these two arrays, the algorithm then generates the pseudo-random sequence. The final loop in Figure 1 shows how this is done; each iteration of the algorithm generates one byte of the pseudo-random stream. This stream can then be XORed with a message to encrypt or decrypt it.

Note that the algorithm described in Figure 1 is very well-suited for software implementation — the algorithm is short and simple, and the data operations (8-bit addition, accessing a small array) are readily available on any processor. In contrast, the extensive sequential data dependencies in the algorithm, especially the second loop that shuffles the S -array depending on the key, complicate a hardware implementation. Nonetheless, the next section will show that significant performance improvements are still possible by implementing the algorithm in hardware.

3 Circuit Design

This section describes the design of the hardware implementation of the RC4 Cracking System.

3.1 Structural Description

Figure 2 shows a block diagram of the circuit. The circuit consists of one main control unit and five functional units. Each functional unit is used to test a different set of keys (*key space*). A block diagram of a functional unit can be seen in Figure 3 (all functional units are identical). Each functional unit contains memory elements for the S - and K -arrays and the i and j counters, along with two adders and one comparator.

3.2 Operational Description

Figure 4 shows the state diagram. The following subsections show how the state diagram and functional unit

correspond to the algorithm in Figure 1.

3.2.1 Initialization of the S-Array

The first step in the initialization of the S -array is the **Load Ram** state where the RAM is filled linearly with $S_0=0, S_1=1$ on up to $S_{255}=255$. This is accomplished by using the i -counter as the data source to the RAM and incrementing the counter every clock cycle.

The second step in the initialization corresponds to “randomizing” the S -array. The algorithm requires that S_i and S_j be read from memory and then written back to memory after a computation. As the RAM blocks in our target CPLD supports only a single read or write access, two reads and two writes are necessary for each of the 256 iterations. These reads and writes to the RAM define the minimum number of clock cycles possible with algorithm and hence the minimum number of states. Figure 4 shows the four states: **Read S_i , Read S_j , Write S_i to S_j , and Write S_j to S_i** . The order of the two reads is predetermined by the algorithm. The order of the two writes is done to preserve the coherence of data between iterations, as the first clock cycle of the new iteration reads from S_{i+1} whereas the old iteration is writing to S_i .

An extra clock cycle is not necessary to compute j as the value output from the computation of j can be fed back to the address port of the RAM during the same clock cycle. In an attempt to increase the clock frequency, a fifth state was considered to compute j and save the value in a register. It was found, however, that the maximum clock frequency of the circuit was not increased with this extra state.

Reduction in the hardware or clock cycles used in the initialization part of the design was considered in two places: the amount of storage required for the K -array and the necessity of loading the S -array with initial values.

A reduction in the amount of storage for the K -array is possible. Since the key is repeated in the K -array, we can store the key once, and use modulo addressing to select a byte when accessing the K -array. This reduces the amount of logic our circuit uses with a minimal increase in complexity.

A reduction in execution time is possible by eliminating the RAM initialization. In the original design, this initialization took 256 clock cycles. This initialization can be eliminated by maintaining a 256-bit register which tracks whether or not each location in a RAM has been updated. During a read, if the register indicates that the location has not been updated, the address can be routed to the output instead of the contents of the RAM. This alternative design was implemented, but it was found that the register consumed a significant portion of the

```

for  $i = 0$  to 255 {
     $S_i = i$ 
     $K_i = \text{key}(i \bmod \text{key\_length})$ 
}
 $j = 0$ 
for  $i = 0$  to 255 {
     $j = (j + S_i + K_i) \bmod 256$ 
    swap  $S_i$  and  $S_j$ 
}
 $i=j=0$ 
for each pseudo-random byte to be generated {
     $i = (i + 1) \bmod 256$ 
     $j = (j + S_i) \bmod 256$ 
    swap  $S_i$  and  $S_j$ 
     $t = (S_i + S_j) \bmod 256$ 
    pseudo-random byte is  $S_t$ 
}

```

Figure 1: RC4 algorithm for generating pseudo-random sequence

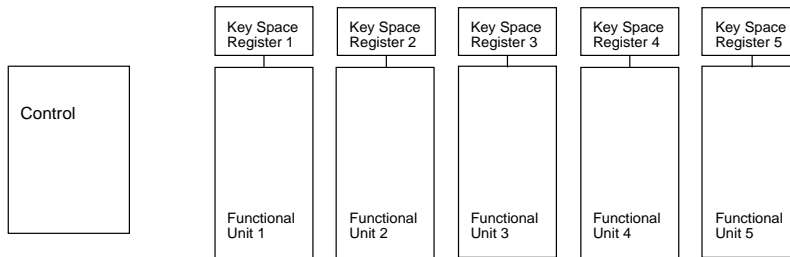


Figure 2: Block diagram of circuit

space on the CPLD, meaning only one functional unit could fit on the device. Without this enhancement, five functional units could fit on the device; thus the enhancement was not used the final design.

3.2.2 Generation of Pseudo-Random Stream

Once the arrays have been initialized, each byte in the pseudo-random stream can be generated, and used to compare one byte of the original message.

The generation and test of the encrypted bytes is very similar to the initialization of the S -array; thus, much of the hardware can be re-used. In each iteration, the values S_i and S_j are swapped. This takes four cycles, and is done in states **In Testing - Read S_i** , **In Testing - Read S_j** , **In Testing-Write S_i to S_j** , and **In Testing - Write S_j to S_i** . During each iteration, a byte in the pseudo-random stream must also be generated; this is done in State **Read S_k** . An additional state, **Clear Test**, is used to initialize the j counter (i need not be initialized, because it will be 0 after the randomization

phase).

Our implementation generates the first four bytes of the pseudo-random stream and uses these to determine whether the correct key has been found. The first, second, and third bytes are tested during the **In Testing - Read S_i** state; the fourth byte is tested during the **Test Done** state. When a correct key has been found, the circuit notifies the user, and enters the *Done* state. Additional logic has been provided to allow the user to probe the state of the machine upon completion, and determine which key resulted in a match.

4 Measured Results

The circuit was implemented on a single Altera EPF10K20 Complex Programmable Logic Device (CPLD) on an Altera UP1 Education Board [2]. The EPF10K20 device was large enough to contain the control unit and five functional units. The S -arrays were implemented using embedded memory arrays, while the K -array and remaining logic were implemented using the

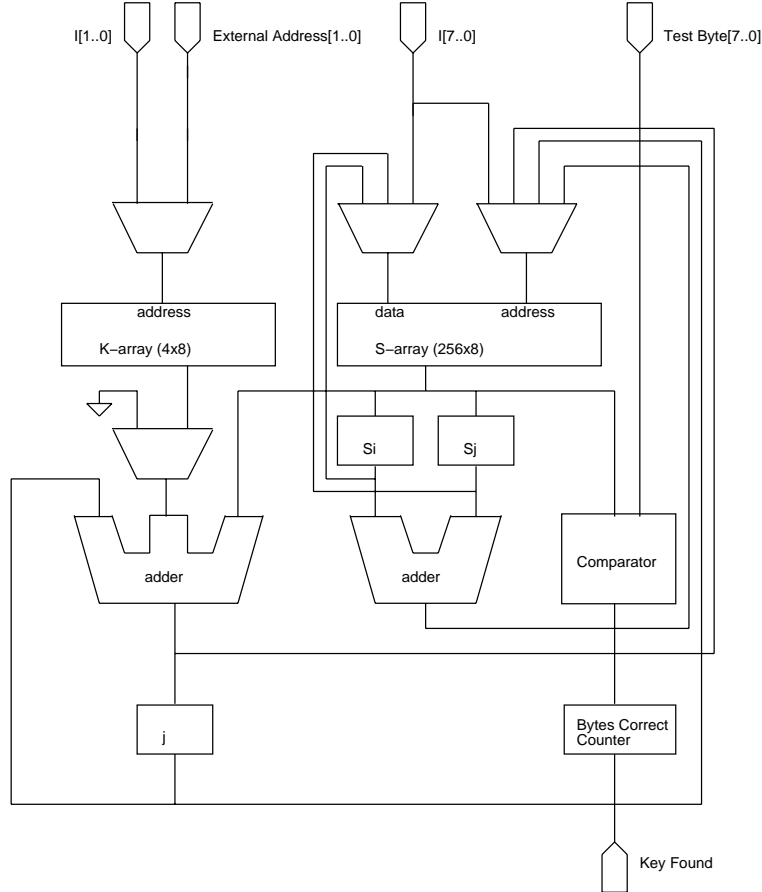


Figure 3: Block diagram of one functional unit

logic portion of the device.

According to the MAX+plusII timing analyzer, the maximum overall clock frequency at which the circuit can be run is 10.38Mhz (we tested the chip at 5Mhz). Overall, the architecture requires 1304 cycles per key; thus, each functional unit can test approximately 8000 keys per second. Since our implementation contains five functional units, our total throughput is 40,000 keys per second. This leads to an expected time of 15 hours to crack a 32-bit RC4 encryption (30 hours worst case). Clearly, the number of keys that must be tested goes up exponentially as the key length increases; our circuit would take 159 days to crack 40-bit RC4 encryption.

Our architecture is trivially scalable. If additional CPLDs or a larger CPLD is available, one can simply add more functional units, thereby reducing the size of the key space that each functional unit must search. Table 1 shows the estimated cracking time for several large CPLDs [2]. The limiting resource in these larger CPLDs is the embedded RAM. In the EPF10K100, there are enough logic elements to implement 23 functional units. However, because the device has only 12 embedded ar-

rays, only 12 S -arrays can be implemented, meaning only 12 functional units can be implemented. The final line in the table corresponds to the 16-board Transmogripher-2 being developed at the University of Toronto (this device will contain two EPF10K100's per board) [3].

For comparison, we also implemented RC4 in software. Our rather straightforward implementation in C, running on a 200Mhz Pentium Pro processor with 256KB cache, was able to achieve a testing rate of less than 22500 keys/second, which gives expected cracking times of 26.5 hours and 283 days for 32-bit and 40-bit keys.

Although careful coding could undoubtedly give much better performance, analyzing the data dependencies in the RC4 algorithm shows the limits of what can be obtained in software. The key loop is the second loop in Figure 1, which scrambles the S -array. Even if we assume an ideal processor that allows multi-ported memory accesses and maximum instructional-level parallelism, we would still require one clock cycle to load S_i and K_i , two more cycles for the two additions, and two more cycles for the load and two stores for the swap. This means that this loop alone will consume $256 \times 5 = 1280$

Device	Number of Functional Units	Cracking Time (32 bits)	Cracking Time (40 bits)
EPF10K20	5	15 hours	159 days
EPF10K100	12	6.2 hours	66 days
EPF10K200	24	3.1 hours	33 days
Transmogriifier-2	384	11.7 min.	50 hours

Table 1: Estimated cracking time for various devices

clock cycles under ideal conditions. Thus, with a 500Mhz processor (and fast enough cache to keep up), we could not possibly test more than 400,000 keys per second — about ten times the rate that we can achieve with a single EPF10K20, for certainly more than ten times the cost. Another way to view this analysis is to note that because of the data dependencies, our CPLD solution uses essentially the minimum possible number of clock cycles per key, and the software solution gains only because of the higher clock frequency.

5 Conclusions and Future Work

This paper has presented a CPLD-based architecture for cracking the RC4 encryption algorithm. On an Altera EPF10K20 CPLD, the circuit can process 40,000 keys per second, meaning we can crack 32-bit RC4 in an expected time of 15 hours. Since the algorithm is trivially parallelizable, we could reach any desired performance level by using more or larger CPLDs.

The primary advantage of our system over other key cracking systems is its low cost. The market value of an EPF10K20 CPLD is approximately CAD\$90, which gives a price/performance of CAD\$0.0047 dollars/key/second. This price/performance does not change significantly as we expand the system by adding more CPLDs. Not only is the component cost low, but the start-up cost to build a minimal system is also quite low, as is the granularity by which we can expand the system. By comparison, our software implementation of the algorithm running on a 200Mhz Intel Pentium Pro can process 22500 keys/second, leading to a projected current price/performance of worse than CAD\$0.0600 dollars/key/second — more than 12 times worse. A custom ASIC implementation could offer much higher performance, but would have prohibitively high start-up costs. Furthermore, the programmable logic solution is reusable for different applications, leading some to dub programmable logic a “Universal Key-Search Machine” [4].

One of the limiting factors in our design was the memory architecture of the CPLD. For the EPF10K20 implementation, the six available embedded arrays were

sufficient to implement the five functional units. For larger devices, however, more memory arrays would be required. In addition, the arrays on our part were all single-port (one memory access at a time). Newer devices have true dual-port arrays [5, 6]; these arrays would have significantly reduced the number of cycles required by our algorithm. In light of these observations, we are currently investigating enhanced memory architectures for CPLDs and FPGAs, and how these enhanced architectures can be used by systems such as the RC4 cracker described in this paper.

Acknowledgements

The authors wish to thank Altera for supplying the UP1 Education Board and the MAX+plus II software.

References

- [1] B. Schneier, *Applied Cryptography*. John Wiley & Sons, second ed., 1996.
- [2] Altera Corporation, *Datasheet: FLEX 10K Embedded Programmable Logic Family*, May 1998.
- [3] D. Lewis, D. Galloway, M. van Ierssel, J. Rose, and P. Chow, “The Transmogriifier-2: A 1 million gate rapid prototyping system,” in *ACM International Symposium on Field-Programmable Gate Arrays*, pp. 53–61, Feb 1997.
- [4] J.-P. Kaps and C. Paar, “Fast DES implementation for FPGAs and its application to a universal key-search machine,” in *5th Workshop on Selected Areas in Cryptography (SAC’98)*, August 1998.
- [5] Altera Corporation, *Datasheet: FLEX 10KE Embedded Programmable Logic Family*, August 1998.
- [6] Xilinx, Inc., “Virtex: Our new million-gate 100-MHz FPGA technology.” *XCell: The Quarterly Journal for Xilinx Programmable Logic Users*, First Quarter 1998.

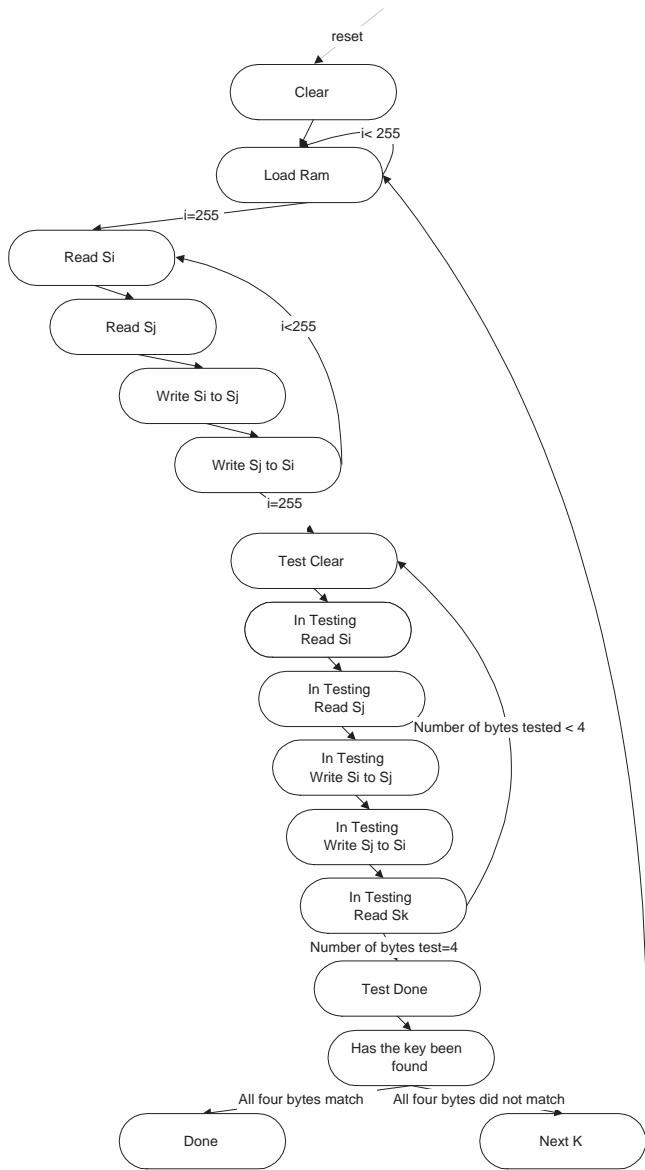


Figure 4: State diagram