Pipelined Data Encryption Standard (DES) Brute Force Attack Unit

S. Foucher, S. Sadhukha and Y. Ton-That
Department of Electrical and Computer Engineering, McGill University, Montreal, Canada
{simon.foucher, shajib.sadhukha, yvan.tonthat}@mail.mcgill.ca

***Abstract*—Brute force cryptic attacks is the only cryptographic approach which guarantees a positive result, provided enough resources are available. Since DES uses block cipher encryption instead of arithmetic algorithms, modern developments in high performance FPGAs have made it a feasible target for brute force attacks because of its short 56 bit key. We developed a system capable of running such an attack on a chosen subset of keys within a reasonable amount of time. By pipelining the 18 DES decryption stages, we implemented 6 decryption unit working in parallel on a single Cyclone II FPGA board capable of running 0.84 billion keys per second at 130MHz. By matching the outputted decrypted block of 64 bits with prior knowledge of the original data encrypted, we were able to generate a positive key match in 9.7 days on average. Even though this might seem a bit slow, we also designed a software equivalent of our device which would require thousands of years to perform the same task, due to the linear arithmetic based processing of regular PC architecture. [Word count = 3244]**

1. INTRODUCTION

THE Data Encryption Standard (DES) was released in 1974 by IBM to serve as a secure cryptographic standard to protect confidentiality of ATM communications. Its ease of implementation in hardware, thought to be one of DES's greatest strength ended up being one of its most significant vulnerability by means of hardware driven brute force cryptic attacks [3]. Unlike arithmetic algorithms, the block ciphers that perform DES encryption can easily be implemented on FPGAs. By pipelining the 18 stages of encryption we have developed a design that can run at 140MHz top speed on an Altera Cyclone II FPGA board. To enhance the feasibility of our design, we have restrained ourselves to a subset of the 56 bit key space containing only alpha numeric ASCII characters, which reduced the scanning task by a factor of 21,500. The motivation for that choice of subset space is an attempt to exploit human laziness in key selection for encryption. At each key iteration, the 64 bit cipher text provided is decrypted and the resulting data is matched with expected value of the plaintext using a lookup table. Our compact architecture enabled us to fit six such devices into a single FPGA, which are capable of processing 0.84 billion keys per second and perform a full attack in three days. Even though this might seem a bit slow, by developing and testing a C# implementation of DES encryption, we found that a software equivalent would require 20,770 years to perform the same task

2. ATTACK UNIT COMPONENTS

To decrypt a 64-bit block of data using the DES algorithm, we used the following basic building blocks: an Inverse Key Scheduling unit, a Decrypter System, a Decryption Unit, a Key Generator unit, and a Look-Up Table (LUT). We were also able to easily design an encryption system by building a regular Key Scheduling unit, which when integrated in the Decryption unit, transforms it into an Encryption Unit (Because of that symmetry in encryption/decryption, the terms 'Encryption Unit' and 'Decryption Units' might be interchanged in this paper, but refer to the same circuit block. The only difference between and Encrypter and a Decrypter is the Key Scheduler which provides the sub keys used. The regular key scheduler is used when encrypting, and the reverse key scheduler when decrypting).

2.1. Key Scheduler

Figure 1 presents the structure of the key scheduler, which accepts a 56-bit key input and through a series of permutations, shifts and re-combinations outputs sixteen 48-bit keys that are to be used to encrypt the message data.

The original key is actually 64 bits in size, but 8 of those bits are parity bits which are only of significance during the transmission of the key. The key scheduler outputs 16 sub-keys K_1 to K_{16}. Those sub-keys are defined as:

$$K_n = FS(n, KEY) \qquad (1)$$

where KS represents a function which takes an integer in the range from 1 to 16 and a KEY as inputs, and outputs a sub-key K_n. [1]

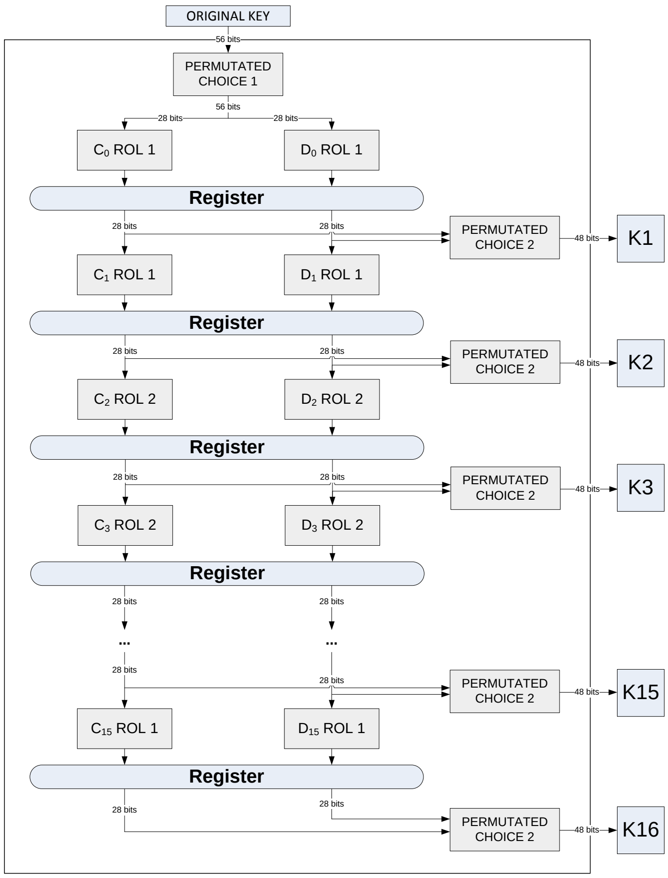

Fig. 1. Structure of the key scheduler

The 56 bits of the original key are fed to the key scheduler's first permutation unit (Permuted_Choice1.vhd), which selects specific bits from the 56 bits of the input key and re-arranges them into two output streams (C_0 and D_0) of 28 bit length each. Each output stream is then left-shifted either 1 or two places respectively (depending on the stage) by the "shift by 1" component (ShiftLeftBy1.vhd) or the "shift by 2" component (ShiftLeftBy2.vhd). The shifted outputs are then recombined through the second key permutation unit (Permuted_Choice2.vhd), the output of which is the stage 1 sub-key. The process of left-shifting (by one or two places as appropriate) and recombining is then repeated 15 times for a total of 16 key stages.

2.2. Encrypter System

Once the key scheduling has been performed, the next step is to prepare the original data block for the actual encryption. This is done by passing the data block through a permutation called the Initial Permutation. This permutation also has an inverse, called the Final Permutation, and it is used in the final stage.

Once the original data block preparation has been completed, the actual encryption is performed by the main DES algorithm through intricate key-dependent computations. The 64-bit block of input data is first split into two halves. The core algorithm of the encrypter, performed by the encryption units, is then applied to those halves 16 times, making up the 16 rounds of standard DES. The high-level view of the encrypter is shown on the right in Figure 2.

The final consideration with the encrypter was to ensure that the sub-keys are applied to the middle stages in order, from K_1 and K_{16}.

2.3. Encryption Unit

Figure 3 illustrates the structure of the encryption unit used in the middle pipeline stages of the encrypter. The purpose of that unit is to perform key-dependent computations via the Feistel function f(R, K).

Once the input data is split into two halves of 32 bits, denoted as L and R, the Feistel function operates on R. Its structure consists of four stages:

1) *Expansion* – the 32-bit half-block is expanded to 48 bits using the expansion permutation. The extra bits are provided by duplicating some of the bits.
2) *Key mixing* – the result from the expansion stage is combined with a sub-key using an XOR operation.
3) *Substitution* − after mixing in the sub-key, the block is divided into eight 6-bit pieces before processing by the substitution box. The substitution box then replaces each of its six input bits with four output bits according to a non-linear transformation, provided in the form of a lookup table. The substitution box provides the core of the security of DES. Without the substitution box, the cipher would be linear and trivially breakable [3].
4) *Permutation* − finally, the 32 output bits from the substitution stage are rearranged according to a fixed permutation.

The Feistel function is always applied to R and the result is then combined with L using an XOR operation. The result is finally stored as the next stage's R. The next stage's L is simply the previous stage's R. Those operations can be defined as follows:

$$L' = R \quad (2)$$

$$R' = L \; xor \; f(R, K) \quad (3)$$

This process is repeated 16 times, making up the 16 rounds of standard DES. Finally, in the final stage the order of the blocks is switched before they are recombined through the final permutation.

2.4. Inverse Key Scheduler

Figure 4 illustrates the structure of the inverse key scheduler. It is nearly identical to the original key scheduler in every regard, the sole difference being the order in which the sub keys are generated [1]. In the original key scheduler, the sub-keys were generated through a combination of left shifts and permutations. In the inverse key scheduler, since the last sub-key K_{16} is actually needed first, it must be generated first. Thus, the total number of left shifts applied to each half of the

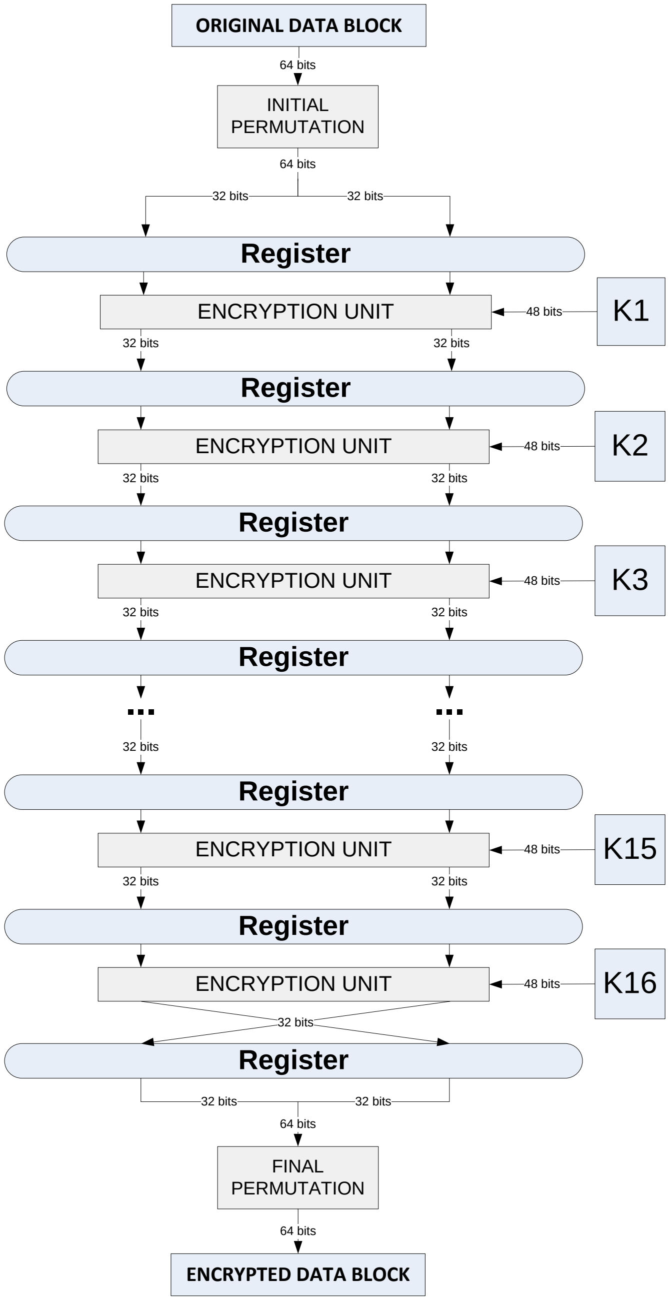

Fig. 2. Structure of the encrypter

input key (C_0 and D_0) from all sub-stages were added up to determine the shift number necessary to produce K_{16}. So at the first stage of the inverse key scheduler, C_0 and D_0 must both be shifted 28 times to the left (actually since C_0 and D_0 are both 28 bits long this amounts to doing nothing). For each subsequent stage, instead of performing a large amount of left shifts, the number by which to shift left is subtracted from 28 to obtain the equivalent number of right shift. As an example, the second stage of the Inverse Key Scheduler needs the input key left shifted by 28:

$$(28_{\text{Data Size}} - 26_{\text{Left Shift}}) = 2_{\text{Equivalent Right Shift}} \quad (4)$$

For all other aspects of the Inverse Key Scheduler, please refer the section for the original Key Scheduler.

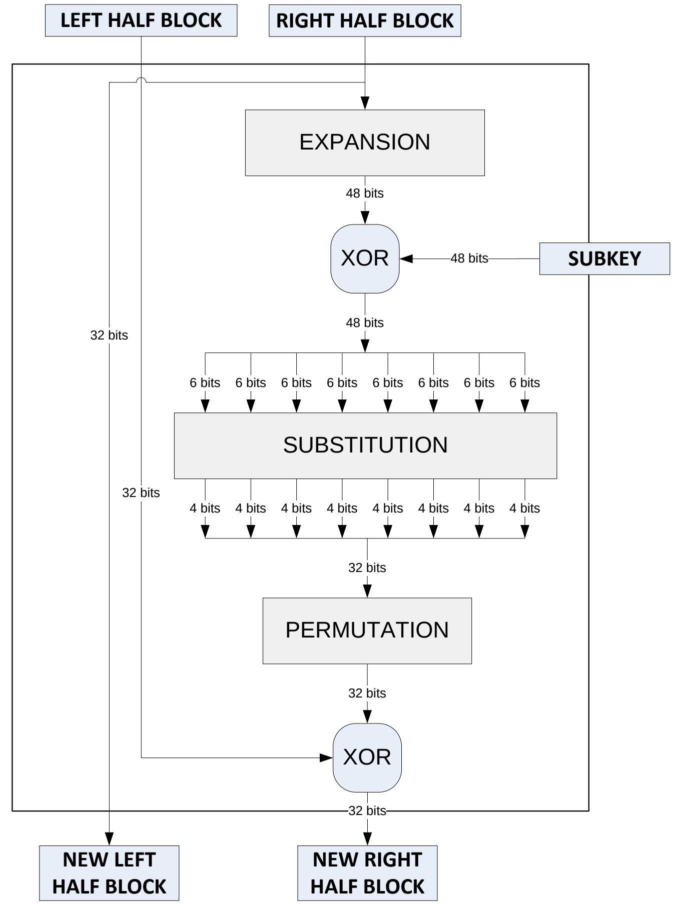

Fig. 3. Internal structure of the encryption unit

2.5. Decrypter System

The same algorithm is used for encryption and decryption. As such, to build the decrypter we simply use the structure of the encrypter as illustrated in Figure 2. The sole different is that the sub-keys are applied in reverse order, from K_{16} to K_1. The structure of the decrypter is presented in Figure 5. [1]

2.6. Key Generator Unit

Note: in the following section, we will use the word 'digit' to represent a 7 bit alphanumeric ASCII character.

The key generator unit takes care of generating the keys which are used to conduct the attack. Since the entire 2^{56} key space was an unfeasible task for our project, we restrained ourselves to a subset of keys containing only ASCII alphanumeric. The unit acts like an 8 digit modulo 62 counter, which outputs letters a-z, then A-Z, then numbers 0-9 in 7-bit ASCII format (8 ASCII characters at 7 bits each gives us the required 56 bit key). Instead of having six key generators (one for every decrypter), we used a single component which outputs six distinct keys per clock cycle. The primary motivations for this architectural choice were first, a reduction in hardware space by eliminating redundancies and secondly, a centralize unit with ease of future modification to

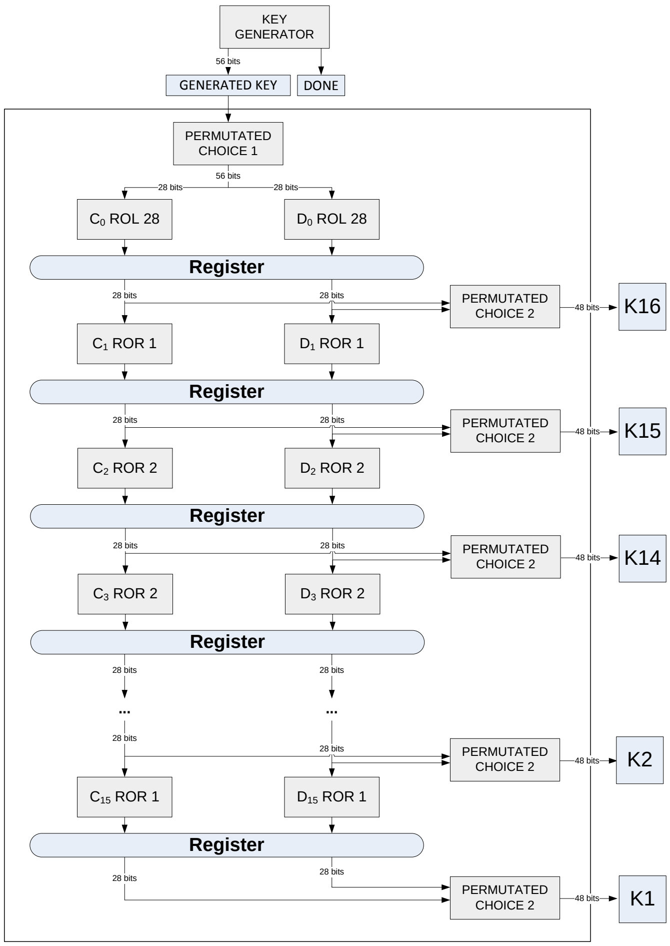

Fig. 4. Structure of the inverse key scheduler

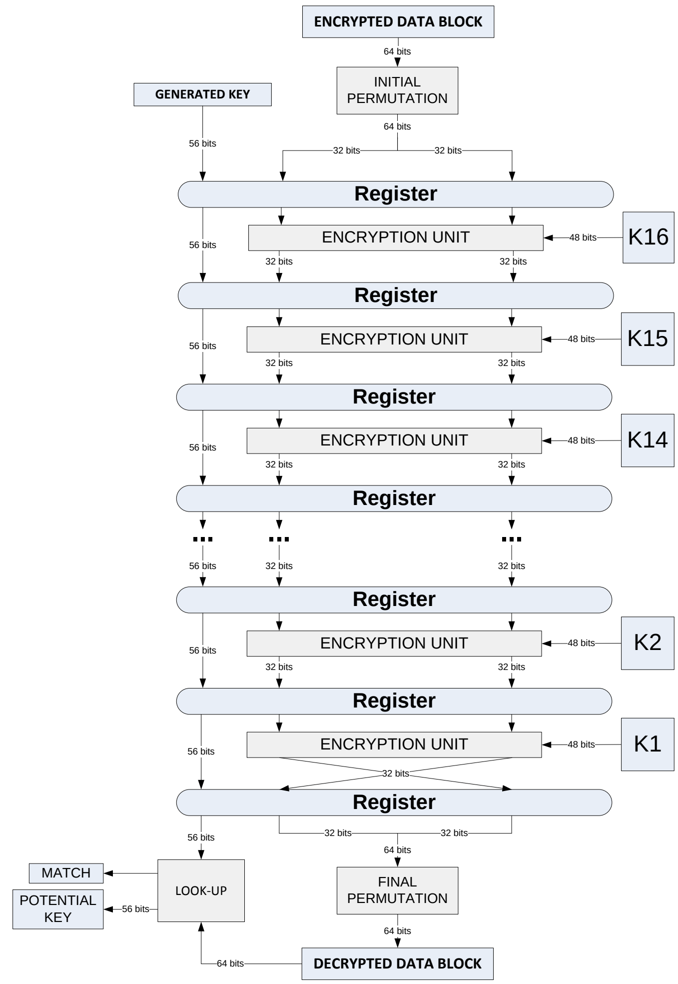

Fig. 5. Structure of the decrypter system

accommodate more decrypter units if implemented in larger boards. The chosen architecture exploits the redundancies in having six key generators by splitting the counter into two parts: the 7 least significant digits of the key (the common key) and the single most significant digit of the key (the specific key). While the common key is the same for every six keys outputted on the rising edge of the clock, the specific byte is different for every key generated. For example, key1 will have a specific key ranging from ‘a’ to ‘k’, key2’s specific key will range from ‘l’ to ‘v’, etc… The common key ranges from ‘aaaaaaa’ to ‘9999999’ and gets incremented at every clock pulse. By appending every individual specific key with the common key, we can generate 6 distinct keys at every clock pulse. Following this example, key1 will start at ‘aaaaaaaa’, while key2 is at ‘laaaaaaa’, all the way to key6 which starts at ‘0aaaaaaa’. At the next clock pulse, key1 will be aaaaaaab’, key2 will be ‘laaaaaab’, etc… The scan will be complete when key1 reaches ‘k9999999’, key2 reaches ‘v9999999’, key6 reaches ‘99999999’, etc… (Note that only alphanumeric characters are scanned, so there is a jump from ‘z’ to ‘A’, from ‘Z’ to ‘0’, and a resetting jump from ‘9’ back to the starting point ‘a’)

With minimal modifications, this design could easily accommodate up to 62 keys/clock cycle by simply splitting the load of the six current specific keys. At 62 keys/clock, every specific key would be hard wired to a single alphanumeric character (a-z, A-Z and 0-9). We could also easily modify the key generator to make it scan all the 56-bit key space in order to perform a full brute force attack.

It is also worth a mention that the system’s critical path resides within one of these transitions, which would be very difficult to pipeline.

2.7. Lookup Table

The basic principle of a brute force attack is to try and decrypt a message with every possible key combination, thereby ensuring a positive result. Once the encrypted data has been decrypted with a given key, a secondary component validates this data to look for coherence. One could use character frequency analysis matched with the sender's language, a dictionary lookup component, or any previous knowledge of the encrypted message. (For example, during WWII, the allies would match the final characters of encoded messages with "Hail Furor").

For our purpose, we implemented a lookup component which matches the deciphered data with a user inputted 64-bit vector. After each attempt to decrypt, this component will compare the output of all 6 decrypters with the user supplied 64-bit vector of expected data and send out a flag signal whenever a match is found, as well as the key used to generate the match.

2.8. Decrypter Nest

The Decrypter Nest is a central management unit controlling several decrypters (six in our case, because of hardware mapping restriction), all working in parallel on a subset of keys provided by the key generator. The decrypter nest receives the first block of 64-bit encrypted message data to be decrypted (either via I/O or hard wired into the VHLD code). It transmits this vector to every decrypter unit which all performs a DES decryption using keys provided by the key generator. After the process and lookup check, the decrypters send their results back to the Nest (match and the used key in that case). The match lines are ORed from the decrypters into the nest's output port, and the output key used are multiplexed (As discussed in the analysis section of this report, it was statistically unnecessary to make provision in the advent of having two units finding a match simultaneously). The Nest has three informative output ports: "PotentialKeyFound" which gets asserted when a match is found; "PotentialKey" which outputs the potential key used to generate the previous flag; and "Finished", which indicates that the key generator has exhausted all its list of keys.

3. PIPELINING

In order to speed-up our brute force search, the decrypter has been pipelined into 18 stages such that it can process one key per clock pulse. Since pipelining only increases the throughput of the system when processing a stream of data, it doesn't make sense to pipeline the encrypter. However, because we are using the exact same structure for both encryption and decryption, we decided to pipeline our encrypter and then use its code to implement the decrypter used in the nest to perform the attack.

Registers were inserted in-between each pipeline stage and clocked synchronously. The time between each clock signal was set to be greater than the longest delay between all stages such that when the registers are clocked, the data that is written to them is the final result of the previous stage. (To our surprise, the critical path actually happened to reside within the key generator, which cannot be pipelined). The logic within the stages is, therefore, purely asynchronous.

The key scheduler was also designed using a pipelined configuration. A register was placed immediately after every shift unit, making up for the 17 pipeline stages of the key scheduler. We also had to ensure that the key scheduler and encrypter (or the inverse key scheduler and decrypter) were properly synchronized such that a sub-key is immediately available at each decryption stage.

4. DISCUSSION

4.1. Hardware Implementation

The development of our components was made and debugged using Model Sim SE V5.8 and the 'Place and Route' analysis, the .sof file generation as well as hardware implementation was done using Altera Quartus II V8.0. Our design was successfully implemented on a Cyclone II (EP2C35F672C6) FPGA board. Even though our design is capable of running at a maximum clock speed of 130MHz (140MHz without the LED display), the target hardware chosen only provides us with a 50MHz clock, thereby degrading our performance by a factor of three.

In order to minimize hardware overhead, the outputted key was transmitted to a basic decoder which translated it to LED signals sent to the board's eight segments LED display. Since it was impossible to distinguish between capital and small letters with the basic eight segments (for example O, o and 0), a single mono sized alphabet was coded using a lookup table, and bit 6 of every ASCII character (1=small letter, 0 = capital) was inverted and sent out to a LED light corresponding to a LED segment decoder. (Therefore, when reading the key, a lit up light would indicate that that character was a capital letter)

Since every individual decryption units took up 15% of the target hardware's resources, we decided to implement 6 of them in the nest, which would total 90% of resources, allowing 10% for overhead. Because of great compiler technology and redundancies in our design, the full unit (including the entire user interface overhead) took only 80% of available logic elements (23,725 Combinational Functions and 7,840 dedicated logic registers). Since at this level of occupancy, the Trace and Route complexity starts to increase exponentially, we were not able to fit a 7th decrypter into the nest.

4.2. Performance

Since the Cyclone II board only provided us with a 50MHz clock, our design was forced to run slower than its maximum potential. At this speed, our system can run the entire chosen subset key space on average in 25 days (50 days worst case).

At maximum clock frequency, this time is reduced to 8.9 days. Even if this might seem a little slow, it is worth mentioning that we are able to process 0.84 billion keys/second, so we could recycle our machine to encrypt data at a rate of 53.7MB/second. We also developed a C# software version of our device, and a comparison in performance is presented in the following table (note that the C# code was not optimized for performance):

TABLE 1
Speedup gained by Hardware versus Software

	Dual Core 1.7 GHz PC	140 MHz FPGA
Keys/Sec	333	840,000
Ave. Time/Key	3.00 mS	1.19 nS
Ave. time/Attack	7,581,255 Days (or 20,770 Years)	8.9 Days

The most widely known commercial alternative for DES brute force attacks is the Copacabana machine (abbreviation of cost-optimized parallel code breaker). This device is available for about $10,000 and, with a cluster of 120 FPGA cores, is capable of running an exhaustive 56-bit search in a matter of weeks. [4]

4.3. Future improvements

The possibility of more than one decrypter in the nest finding a key match at the same time did not escape to us. Based on the Birthday Paradox, the number of inputs to a Hashing function required to generate a hash collision is given by $2^{(\text{output bits}/2)}$ [5]. Using this as an upper bound (the DES is a Hash table optimized to avoid differential analysis, so that these collisions are minimized) we can estimate that the biggest number of keys which will generate the same output is 2^{32}, which on a key space of 2^{56} gives the probability of a collision $p = 5.96*10^{-8}$. Using a binomial expansion, we can conclude that having two collisions when running six decrypters has a probability of $(6C2)*p^2*(1-p)^4 = 5.32810^{-14}$. Based on this extremely low probability, which is an upper bound, no provisions were made to account for that possibility.

This also tells us that the upper bound expected number of potential keys found can be given by $E(x) = n*p$, where $n = 62^8$ (the number of keys we are exploring), which is just about 13 million. (Note once again that this is an upper bound for linear hash functions and that DES is a non linear system optimized to reduce this number). At the present moment, our system is incapable of dealing with such a large number of keys. A good improvement would be to feed those keys to a second stage decrypter which could then use either more knowledge on the encrypted text, or run various data coherency tests (i.e. dictionary lookups, ASCII lookups, etc.). This improvement is presented in Figure 6. The expected number of potential keys found on that second level of decrypter would be 0.77, so entering that second stage with knowledge that the message was encoded by one of the presented keys would statistically ensure that a single match is found.

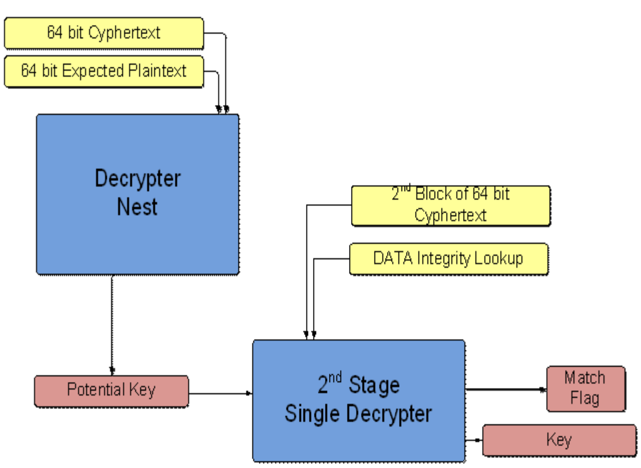

Fig. 6. Future improvement: a second stage device

5. CONCLUSION

By pipelining the DES decrypter and running six of them in parallel, we were able to speed-up our device by a factor of 108 (as compared to a device with a single non-pipelined version of the decrypter). This enabled us to run through our subset of keys in a reasonable amount of time. Since the entire 56-bit DES key space contains 330 times more entries than what we have tested, a full brute force attack would either require a few years of processing, or many FPGAs in parallel.

In any case, the speedup observed in the hardware implementation of DES is significant enough to rule out any software equivalent to perform these kinds of tasks. As the power of FPGAs increase, so do the strength of ciphers and the length of keys used. The DES has now been replaced by the Triple DES (Triple Data Encryption Algorithm-TDEA) which uses the same hardware as DES, but encrypts a block of data three times with three different keys. A brute force on a 168-bit key is theoretically impossible, and even with modern FPGAs it would take more than the age of the universe to perform a full brute force search. A better approach to decryption could be to use a more sophisticated attack like differential analysis, which could reduce the complexity of the task by a very large factor, or simply hack into the sender's computer network and access data directly from the source.

REFERENCES

[1] *Data Encryption Standard*, U.S. DEPARTMENT OF COMMERCE/ National Institute of Standards and Technology, FIPS PUB 46-3, Oct. 1999.

[2] H. Yang. (2007). Cryptography Tutorials - Herong's Tutorial Notes [Online]. Available: http://www.herongyang.com/crypto/des_implTest .htm

[3] E. Biham and A. Shamir, "Differential Cryptanalysis of DES-like Cryptosystems," Dept. Applied Mathematics, The Weizmann Institute of Science, Jul. 1990.

[4] *COPACOBANA: A Codebreaker for DES and other Ciphers*, COPOCOBANA [Online]. Available: http://www.copacobana.org

[5] *Birthday Problem*, Wikipedia online [Online]. Available: http:// en.wikipedia.org/wiki/birthday_paradox