**Pipelined DES Brute Force Attack Unit**

*Shajib Sadhukha (*260090625*),* *Yvan Ton-That(260188680), Simon Foucher (2602223197)*

#### Abstract

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. 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 21 000. 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 6 such devices into a single FPGA, which are capable of processing 0.84 billion keys per second. To evaluate the hardware advantage, we tested a software implementation of DES encryption and found a speedup by a factor of !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!.

**1. Introduction**

**2. Attack Unit Components**

To encrypt or decrypt a 64-bit block of data using the DES algorithm, our final project will need the following basic building blocks: a Key Scheduling unit, an Encryption unit, an Inverse Key Scheduling unit, a Key Generator unit and an LUT. Note a Decryption unit is not necessary since decryption is accomplished by using the same Encryption unit fed by the Inver Key Scheduler unit. The only difference being the order in which the keys are fed into the system (in reverse order for decryption). The following sections will offer insight into each one of the units mentioned above.

**2.1. Key Scheduler**

The Key Scheduler (Key\_Scheduler.vhd) 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 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 56 real key bits 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 (C0 and D0) 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 subkey. The process of left-shifting (by 1 or 2 places as appropriate) and recombining is then repeated 15 times for a total of 16 key stages.



Figure Key Scheduling Unit

The 16 subkeys K1 to K16 are defined as:

*Kn = KS(n,KEY)*

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 Kn.

**2.2. Encryption Unit**

Given a message to be encrypted, this unit submits the input message data to an Initial Permutation (IP; Initial\_Permutation.vhd), followed by intricate key-dependent computations and ends with a final permutation consisting of the inverse of the initial permutation (IP-1 ; Inverse\_Permutation.vhd).



Figure Encryption Unit

The key-dependent computations are in essence the cipher functions f(R, K) (cipher\_function.vhd) and the keys on which they depend are provided by the key scheduler. A sketch of the calculation of f(R, K) is given in figure 3:



Figure 3 S Functions

The functions in the centre of the diagram are the selection functions S1, S2... S8. They take 6-bit blocks as inputs and yield 4-bit blocks as outputs. Following the IP, the key data is split up into 2 blocks of 32 bits Left (L) and Right (R). The cipher function is always applied to the R block and the result is then added to the L block and finally stored as the next stage’s R block (R'). The next stage’s L block (L') is simply the previous stage’s R block. The stage outputs L' and R' are defined as follows:

*L' = R*

*R' = L(+)f(R,K)*

This process is repeated for 16 stages and in the final stage the order of the blocks is switched before they are recombined through the final permutation IP-1.

**2.3. Inverse Key Scheduler**

The Inverse Key Scheduler (InverseKeySchedule.vhd) is nearly identical to the original Key Scheduler in every regard, the sole difference being the order in which the sub keys are generated. In the original Key Equivalent , the sub keys were generated through a combination of left shifts and permutations. In the Inverse Key Scheduler, since the last sub key (K16) is actually needed first, it must be generated first. Thus, the total number of left shifts applied to each half of the input key (C0 and D0) from all sub stages were added up to determine the shift number necessary to produce K16. So at the first stage of the Inverse Key Scheduler, C0 and D0 must both be shifted 28 times to the left (actually since C0 and D0 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 need the input key left shifted by 28

$$\left(28\_{Data Size}-26\_{Left Shift}\right)= 2\_{Equivalent Right Shift}$$

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



Figure Inverse Key Scheduler

**2.4. Keygen Unit**

The KeyGen 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 bit modulo 62 counter, which outputs letters a-z, then A-Z, then numbers 0-9 in ASCII format. Instead of having 6 keygens (1 for every decrypter), we used a single component which outputs 6 keys. The chosen architecture exploits the redundancies in having 6 keygens by splitting the counter into 2 parts: the 7 least significant bytes of the key (the common key) and the single most significant byte of the key (the specific byte). While the common key is the same for every 6 keys outputted on the rising edge of the clock, the specific byte is different for every key generated. Key01 scans a-k, key02 scans l-v, key03 scans w-F, etc… Therefore, when the 7 byte ‘a to 9’ common counter reaches ‘999999’ for the first time, it resets itself to ‘aaaaaaa’, the specific byte of key01 is incremented from ‘a’ to ‘b’, making key1 transit from ‘a9999999’ to ‘baaaaaaa’, the specific byte of key02 is incremented from ‘l’ to ‘m’ making key02 transit from ‘l9999999’ to ‘maaaaaaa’ and so on and so forth.

 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.

 With the 50MHz limitation of the Cyclone II board, the system can run the full chosen keyspace on average in 25 days (50 days worst case). The only additional speedup we could propose would be to further restrict the sub keyspace, or to link many boards in parallel as does the copacobana machine -abbreviation of cost-optimized parallel code breaker, which conglomerated 120 FPGA cores and is capable of running an exhaustive 56bit search in a matter of weeks).

**2.5. Lookup Table**

The basic principle of a brute force attack is to try and decrypt a message with every possible key combination to assure a positive result. Once the encrypted data has been decrypted with a given key, asecondary 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").

TO BE IMPLEMENTED IN HARDWARE BY SIMON. 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.

**2.6. Decryption Unit**

This unit represents several independent Decrypter systems (Decrypter\_System1…N.vhd) which are managed by the Decrypter Nest (Decrypter\_Nest.vhd). The Decrypter Nest receives the 64 bit encrypted message data vector to be decrypted. It outputs the decrypted message as well as control signals indicating whether or not the decryption was successful. When receiving an encrypted message, the first 64 bits of the cipher text are assumed to be "Dear Gro". Each Decrypter System follows the DES encryption standard to decrypt the data and outputs a matched key once data matching “Dear Gro” is found. The Decrypter Nest has 3 informative output ports: PotentialKeyFound is asserted when a match is found, PotentialKey outputs the potential key and Finished, which indicates that the decrypted has exhausted the key list.

The reasoning behind having 6 independent Decryption Systems is that the board being used for this experiment can accommodate 5 Deccryption Systems. So instead of having a single Decryption System scan through the entire key space, each system is assigned a sixth of this keys pace.



Figure Decrypter

**2. Pipelining**

The decrypter has been pipelined into 18 stages. The first and last stages are the initial and final permutation, and the 16 remaining stages are the encryption stages (XOR of L, R and the 'F' functions) which use a sub key generated by the key scheduler. All the logic in the stages is purely asynchronous, and the timing is achieved by the clocked registers placed between each stage.

 The key scheduler was designed using a pipelined configuration. A register was placed immediately after every shift units (except after the first pair of shift units). This way, the key scheduler has 18 stages, just like the pipelined-version of the decrypter. The timing of the pipelined key scheduler had to be synchronized with that of the decrypter since at each intermediate decryption stage, the corresponding intermediate key must be available for proper decryption.

References

[1] A.B. Smith, “Article Title,” *Journal Name*, Vol. 1, No. 2, pp. 1-10, Month, Year.

[2] C.D. Jones, *Book Title*, Publisher, City, Year.

[3] J. Doe, “Article Title,” in *Name of Conference*, City, Month Year.

[4] I. van Beijnum, “Internet Census: Researchers Knock on 2.8 Billion Virtual Doors,” available online at http://arstechnica.com, October 17, 2007.