Using FPGAs to Parallelize Dictionary Attacks for Password Cracking

Yoginder S. Dandass
Box 9637
Computer Science and Engineering
Mississippi State, MS 39762, USA
yogi@cse.msstate.edu

Abstract

Operating systems and data protection tools are employing sophisticated password derived encryption key techniques in order to encrypt data. Such techniques impose a significant computational burden on forensic tools that attempt dictionary attacks are requiring cryptographic hash generation functions to be called several thousand times for each password attempted.

In order to improve throughput, forensic analysis tools are designed to operate in a distributed manner over a dedicated network of workstations. This paper describes an FPGA-based hardware implementation of the standard PKCS#5 technique published by RSA Laboratories for generating password-derived encryption keys. This is the most computationally demanding step required when performing a dictionary attack on modern password-protected systems.

The initial FPGA implementation incorporates four password-derived encryption key generation units operating at a frequency of 150MHz and is capable of processing over 510 passwords per second. The implementation's performance can be easily improved by incorporating additional key generation units.

1. Introduction

Examining the content of files on disk drives is an essential activity in any digital forensic analysis of a subject computer. However, the easy availability of encryption tools, such as the Microsoft Windows Encrypting File System (EFS), is making this analysis more difficult. Furthermore, the increasing sophistication of encryption schemes employed by such tools is also necessitating the development of more sophisticated encryption "cracking" tools.

In general, *asymmetric* cryptography techniques are significantly more computationally intense than *symmetric* cryptography tools. In symmetric cryptography, a single key is used for encrypting and decrypting data. In asymmetric cryptography, a user's public key is used for encrypting data. The encrypted data can only be decrypted using the user's private key. Typically, the user's public key is *published* (*i.e.*, is not a secret); however, the user's private key is kept secret.

Because asymmetric cryptography is 1000 times slower than symmetric cryptography, file encryption tools use symmetric cryptography to rapidly encrypt and decrypt the file content. However, the symmetric cryptography key – also called the file encryption key (FEK) – used for encrypting the file is itself encrypted using asymmetric cryptography. Because the FEK cannot be determined from the user's public key, forensic investigation requires that the user's private key be determined before the content of encrypted files can be examined.

Cryptography tools typically also protect the user's private keys. In Windows, for example, EFS randomly generates the user's public and private encryption keys (unless they have already been previously created) and then encrypts the private key using a *session* key derived from the *master* key. The Windows master key is a 512 bit random key that is, in turn, symmetrically encrypted with a key derived from the user's password [1].

Using a brute-force attack for decrypting the user's public key is impractical. Instead, a better approach is to use a *dictionary* attack in order to determine the user's password, and thereby, to gain access to all of the user's "secrets" used for data protection. In a typical dictionary attack, a list of potential passwords is generated based on knowledge about the user and a software tool iteratively generates a password-derived symmetric encryption/decryption key that is used for decrypting the user's master key and then uses the decrypted master key for decrypting the user's session keys, private keys, FEKs, and file content.

Generating the password derived encryption key (PDEK) is the most time consuming operation performed by the dictionary attack tool. Also, this operation needs to be performed for every password

1530-1605/08 $25.00 © 2008 IEEE

that is attempted. Therefore, any acceleration of this process has the potential for significantly speeding up a forensic investigation. Commercial digital forensic tool vendors provide tools that parallelize the dictionary attack by distributing the processing over several processing nodes on a *cluster* (*i.e.*, a network of dedicated workstations) [2].

Cluster computing, however, is not necessarily the best method for applying a dictionary attack because each cluster node requires resources (*e.g.*, space and power) that a typical personal computer or workstation requires. The use of field programmable gate array (FPGA) based technology has the potential for significantly speeding up the process that generates the PDEKs with a concomitant decrease in power consumption and space requirements.

This paper presents a technique for using FPGAs for implementing the PDEK generation portion of the dictionary attack toolkit. The remainder of the paper is organized as follows: Section 2 provides background on existing password cracking tools. Section 3 provides details on the PDEK generation techniques and its implementation in an FPGA. Section 4 presents experimental results, and Section 5 concludes with an analysis of the results and avenues for further optimization.

2. Background

This section describes a number of password cracking tools for a variety of operating systems. It also describes a variety of FPAG-based implementations of cryptographic functions.

2.1. Common password cracking approaches

John the Ripper (John) is arguably the most well-known password cracking program [3]. This program uses a dictionary of words in a variety of languages and character sequences most often used in passwords. Based on its sophisticated dictionary, John can quickly be used to break into systems and user accounts protected by weak passwords (*i.e.*, passwords consisting entirely of words in commonly used languages such as English). John can also be used to break passwords that incorporate simple permutations of numerical or special characters in the password words.

In many cryptographic and data protection systems, a cryptographic hash of each user's password is stored along with the users information in order to establish the user's identity when he/she logs in. This is also a mechanism to protect the password because the plaintext password is never stored in the system. When then user enters the password, the login subsystem re-computes the password hash and compares the hash to the hash value stored on disk to confirm password correctness.

However, this simple approach to password protection enabled attackers to pre-generate hashes for a large number of passwords. Although this technique requires large amounts of storage, passwords could be cracked relatively quickly because the attacker would only need to perform a simple lookup to determine the set of passwords that correspond to the hashed password.

Ophcrack is one such tool that uses pre-generated database of hashes and corresponding passwords in order to accelerate password cracking [4]. Instead of maintaining a comprehensive list of all hashes of all passwords, Ophcrack uses the concept of "rainbow tables" in which only a small fraction of all passwords and hashes need to be stored in the pre-generated database, thereby reducing the required storage. The rainbow table technique uses a "reduction" function that produces a new password from a given hash.

Given a starting password and alternatively applying the hashing and reduction functions a chain of several password and hashes is generated. The first and last passwords are the only items required to be stored in the database, all other passwords and hashes can be recomputed from this information. Ophcrack generates and stores a number of such chains in its rainbow table.

Given a hash stored on a hard disk, Ophcrack starts to generate another chain of passwords and hashes beginning at the given hash. If a generated password matches any of the chain ending passwords in the rainbow table, Ophcrack regenerates the corresponding chain using the password that starts the chain in order to determine the password corresponding to the hash on the disk. This ingenious implementation of the "time-memory trade-off," first proposed by Martin Hellman, fails however, just as other hash lookup attacks fail, when a "salt" is used to randomize the has generation.

Most modern PDEK techniques utilize a random salt value as an additional parameter for the hash function. This random value significantly increases the amount of pre-computed data that must be maintained by hash table lookup techniques, rendering them largely ineffective for cracking passwords on newer versions of Windows and Linux.

2.1. Password-based cryptography

PKCS#5 version 2.0 is a standard technique published by RSA Laboratories for generating a

password derived encryption key (PDEK) from a user password, a salt value, and a specified number of iterations [5]. The iteration value (typically larger than 1,000) is used to increase the number of computations required to generate the PDEK. For a legitimate user, a one time computation of 1,000 iterations of a hashing function does not pose a significant computational burden when generating keys or validating passwords. However, for an attacker, computing extra 1,000 hash iterations for every password guess attempt adds significant computational overhead to an exhaustive search algorithm. Also, as described previously, the use of the salt makes simple table-driven lookups for mapping hashes to passwords difficult.

For clarity, the PDEK generation algorithm specified by PKCS#5 version 2.0 for Windows XP is described below as opposed to the general PKCS#5 algorithm.

The PDEK generation algorithm is described as follows:

$$PDEK = PBKDF2\ (P, S, c),$$

where PDEK is the resulting 120-bit (20-byte) string that represents the output password derived encryption key; P is the password string; S is a 16-byte salt string, and $c \geq 4000$ is the number of iterations. The number of iterations is specified by registry value `MasterKeyIterationCount` stored under the `HKEY_LOCAL_MACHINE\Software\Microsoft\Cryptography\Protect\Providers\GUID` typically with a value of 4,000, but this value can be increased by the system administrator.

The *PBKDF2* function essentially performs an exclusive-or of the results from c invocations of the HMAC-SHA-1 algorithm as given by the following recursive expression:

$$PBKDF2(P,S,c) = U_1 \oplus U_2 \oplus ... \oplus U_c,$$

where U_1, U_2, …, and U_c are the results of c invocations of the HMAC-SHA-1 algorithm and $\oplus$ is the exclusive-or operator. The SHA1 algorithm invocations are performed as follows:

$$U_x = \begin{cases} HMAC-SHA-1(P, S \,||\, INT(1)) & \text{when } x = 1 \\ HMAC-SHA-1(P, U_{x-1}) & \text{when } 1 < x \leq c \end{cases}$$

where HMAC-*SHA*-1(k,m) is an invocation of the HMAC-SHA-1 algorithm such that k is the encryption key and m is the text to be encrypted. S||*INT*(1) represents the salt concatenated with the 4-byte binary representation of the value 1 (one) – most significant byte first. Note that the computation of U_x depends on the value of U_{x-1}. This makes it impossible to compute the 4,000 invocations of the HMAC-SHA1 algorithm in parallel.

2.2. The HMAC-SHA-1 algorithm

HMAC-SHA-1 is an algorithm used for verifying the authenticity and integrity of messages using the SHA-1 algorithm as the cryptographic hashing function [6]. The HMAC-SHA-1 function is defined as follows:

$$h(k,m) = sha1((k \oplus opad) \,||\, sha1((k \oplus ipad) \,||\, m))$$

where *opad* is a sequence of 64 identical bytes each containing the binary value 01011100 (or 5C in hexadecimal notation); *ipad* is a sequence of 64 identical bytes each containing the binary value 00110110 (or 36 in hexadecimal notation); $\oplus$ denotes the exclusive-or operator and || denotes the concatenation operator. If the key, k, is larger than 512 bits, then the key is hashed using the SHA-1 algorithm and the hashed key is used as the key for the HMAC-SHA-1 algorithm. However, in this application, the key is the user's password and is assumed to be less than 64 byes. Therefore, this step is not implemented in this research project.

2.3. The SHA-1 algorithm

The SHA-1 algorithm is designed to produce a *message digest* (*i.e.*, a non-reversible hash) for messages of size l where $1 \leq l \leq 2^{64}$ [7]. The SHA1 algorithm divides the message into a sequence of 512-bit non-overlapping blocks labeled $M(1)$, $M(2)$, …, $M(n)$ and applies a series of 80 operations to each block. The hash resulting from the operations after all n message blocks have been processed is the message digest value.

The SHA-1 algorithm consists of the following data elements:

- 80 constants labeled K_0, K_1, …, K_{79} with the following values:

$$K_t = \begin{cases} 5A827999_{16} & \text{where } 0 \leq t \leq 19 \\ 6ED9EBA1_{16} & \text{where } 20 \leq t \leq 39 \\ 8F1BBCDC_{16} & \text{where } 40 \leq t \leq 59 \\ CA62C1D6_{16} & \text{where } 60 \leq t \leq 79 \end{cases}.$$

- Five 32-bit variables labeled A, B, C, D, and E.
- Five 32-bit variables labeled H_0, H_1, H_2, H_3, and H_4.
- 80 32-bit variables labeled W_0, W_1, …, W_{79}.
- Initialize H_0:=67452301_{16}, H_1:=$EFCDAB89_{16}$, H_2:=$98BADCFE_{16}$, H_3:=10325476_{16}, and H_4:=$C3D2E1F0_{16}$.

Before the algorithm begins, the input message is padded in order to produce a total message length that is a multiple of 512 bits. The padding essentially

concatenates the following bit sequence to the original message:
1 <sequence of q 0 bits> <64 bit representation of l>
where l is the original message size in bits and q is given by 447–(l *mod* 512). The algorithm proceeds by processing each 512-bit block $M(i)$ of the padded input message in sequence as follows:

- Divide $M(i)$ into 16 32-bit words W_0, W_1, …, W_{15}, such that W_0 is the left-most word.
- For $16 \le t < 64$, compute $W_t := rotl_1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16})$ where $rotl_1(X)$ is a one-position 32-bit left rotate operation.
- Set $A := H_0$, $B := H_1$, $C := H_2$, $D := H_3$, and $E := H_4$.
- For $0 \le t < 64$, compute $E := D$; $D := C$; $C := rotl_{30}(B)$; $B := A$; and $A := rotl_5(A) + \xi_t(B,C,D) + E + W_t + K_t$ where ξ_t represents a non-linear function described in more detail below and $rotl_{30}$ and $rotl_5$ left-rotate 32-bit input variables by 30 and 5 positions, respectively.
- Compute $H_0 := H_0 + A$, $H_1 := H_1 + B$, $H_2 := H_2 + C$, $H_3 := H_3 + D$, $H_4 := H_4 + E$.

The non-linear function ξ_t is defined as follows:

$$\xi_t(X,Y,Z) = \begin{cases} (X \wedge Y) \vee (\overline{X} \wedge Z) & \text{where } 0 \le t \le 19 \\ X \oplus Y \oplus Z & \text{where } 0 \le t \le 19 \cdot \\ (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z) & \text{where } 0 \le t \le 19 \\ X \oplus Y \oplus Z & \text{where } 0 \le t \le 19 \end{cases}$$

After all $M(i)$ have been processed, the 160-bit (20 byte) sequence formed by $H_0 || H_1 || H_2 || H_3 || H_4$ is the value of the message digest.

A number of FPGA-based implementations of the SHA-1 algorithm have been reported in the literature [8 -10]. The different implementations trade-off the amount of FPGA resources consumed versus the performance of the implementation. Implementations focusing on minimizing FPGA resource consumption implement the SHA-1 using a looping architecture. Implementations focusing on performance typically perform loop unrolling in order to create a design with four pipelined stages such that each pipeline stage operates on a different $M(i)$ in parallel.

Of these, the implementation by Kakaroutas *et al.* is the fastest in terms of clock frequency [8]. Their implementation balances the operations performed in each stage in order to reduce the length of the critical path, resulting in a high-performance implementation.

3. FPGA-based PDEK generation

The implementation of the PDEK generation algorithm developed for this project is based on the SHA-1 implementation by Kakaroutas *et al.* The central part of the FPGA implementation is the repeated invocation of the HMAC-SHA-1 algorithm. HMAC-SHA-1 essentially performs exclusive-or operations, concatenations, and invokes the SHA-1 algorithm twice for each invocation of the HMAC-SHA-1 algorithm. Therefore, optimization of the SHA-1 implementation is the primary factor in improving the throughput of PDEK generation.

Because the implementation of HMAC-SHA-1 and SHA-1 is specifically targeted towards PDEK generation, as opposed to general message authentication and hashing, their design is modified in order to account for short key and message sizes (at most two 512-bit blocks) and to provide a limited level of parallelism in the face of the restrictions imposed by the PKCS#5 algorithm.

The following pseudo code describes the PDEK generation algorithm in terms of SHA-1 invocations, concatenations, and exclusive-or operations:

```
1: PDEK(P,S,c){
2:  U:=SHA1((P⊕opad)‖
            SHA1((P⊕ipad)‖S‖Int(1)))
3:  K:=U
4:  for (i=1; i<c; i++){
5:   U:=SHA1((P⊕opad)‖
             SHA1((P⊕ipad)‖U))
6:   K:=K⊕U;
7:  }
8:  return K;
9 }
```

3.1. Padding operations

In this implementation, passwords are assumed to be no longer than 32 bytes (this restriction will be relaxed in a future implementation). Each password in the dictionary of passwords is represented as a 32 byte string. Passwords that are less than 32 bytes are padded on the right with as many NULL (0-value) bytes as required to create 32-byte strings. Because these passwords are combined with the 64-byte long *opad* and *ipad* strings using the exclusive-or operation in the HPAC-SHA-1 function, this representation is convenient and requires no additional processing overhead. Furthermore, while this password organization scheme is wasteful in terms of storage space required for the dictionary, it does not impose an undue burden because disk capacities are becoming increasingly large. More efficient representation techniques will need to be investigated when the project is expanded to access remote password dictionaries via network connections.

In line 2 of the pseudo code above, the SHA-1 algorithm is called twice. In the first invocation, the message consists of (P⊕ipad)‖S‖Int(1). Recall that (P⊕ipad) is a 64-byte string, S is a 16-byte string, and Int(1) is a 4-byte string. This means that the message length for this invocation of SHA-1 is an 84-byte string.

For the second invocation of the SHA-1 algorithm in line 2, the message is (P⊕opad)||SHA1 where SHA1 is the result from the previous SHA-1 invocation. Again, recall that (P⊕opad) is a 64-byte string and SHA1 is a 20-byte string. This means that the message length for the second SHA-1 invocation is also an 84-byte string

Similarly, in line 5, for the first invocation of SHA-1, the message is (P⊕ipad)‖U where (P⊕ipad) is a 64-byte string and U is the 20-byte result from the previous invocation of SHA-1. This also results in an 84-byte message string. Lastly, for the second SHA-1 call in line 5, the message is (P⊕opad)||SHA1 where SHA1 is the result from the previous SHA-1 invocation. This also results in an 84-byte message string. Therefore, it is clear that the SHA-1 algorithm implementation for the PDEK generation procedure can be specialized to deal with 672-bit input messages.

For a SHA-1 implementation, a fixed message size of 672 bits means that there will be at exactly two 512-bit message blocks (i.e., $M(1)$ and $M(2)$) and that $M(2)$ will need to be padded using the procedure described in section 2.3. However, because $M(2)$ always consists of 160 bits initially, the padding can be accomplished by adding a fixed 352 bit long bit pattern (*i.e.*, no padding-related computation needs to be performed at runtime).

3.2. Pipelined SHA-1 implementation

A SHA-1 implementation essentially needs to perform 80 operations per message block in order to produce the message digest from a single message. The general-purpose implementation designed by Kakaroutas *et al.* organizes the 80 operations into four pipelined stages where each stage performs 20 operations. This architectural organization naturally follows from the specification of the SHA-1 algorithm in which four different non-linear functions, ξ, are utilized. This architecture also enables up to 4 different message blocks to be processed in parallel in the pipeline stages.

However, in the PDEK application, there are only two message blocks in each invocation of SHA-1. This means that for this application, only two pipeline stages are required (it also means that speedup due to pipeline parallelism is half of what is achieved by Kakaroutas' implementation).

In the PDEK implementation, each pipeline stage has a latency of 40 clock cycles. When the message block $M(1)$ enters the second pipeline stage, message block $M(2)$ enters the first pipeline stage. This means that processing $M(1)$ and $M(2)$ through the two pipeline stages requires a total of 120 clock cycles. Stated another way, during clock cycles 1 through 40, $M(1)$ is processed by the first pipeline stage while the second stage is idle. During clock cycles 41 through 80, $M(1)$ is processed by the second pipeline stage and $M(2)$ is processed by the first pipeline stage. At the end of the 80^{th} clock cycle, $M(1)$ processing completes. During clock cycles 81 through 120, $M(2)$ is processed by the second pipeline stage while the first stage is idle. $M(2)$ processing is completed at the end of the 120^{th} clock cycle, resulting in the message digest.

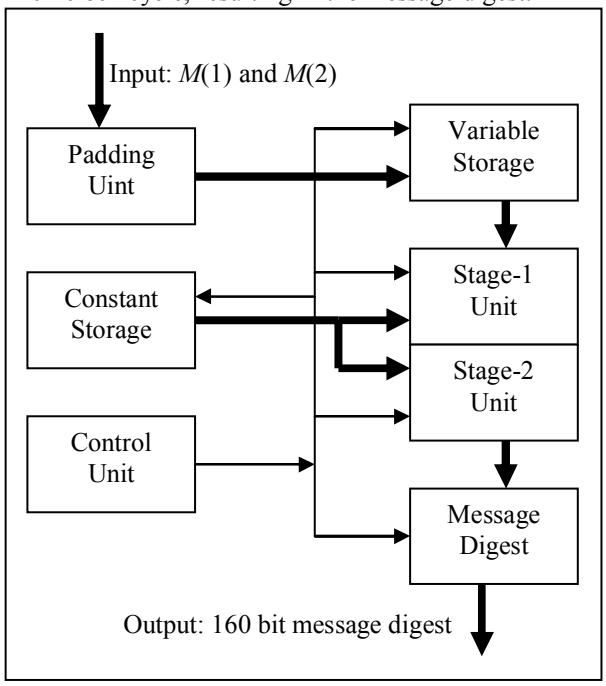

Figure 1: SHA-1 Module Block Diagram

Figure 1, adapted from [8], describes the implementation of the SHA-1 module. The *control* unit consists of a finite state machine with 122 states that controls the operation of the SHA-1 module. The *padding* unit generates the required padding when $M(2)$ is input. The *constant storage* unit stores K_t values. The *variable storage* unit is used for maintaining W_0, W_1, …, W_{79}, A, B, C, D, E, H_0, H_1, …, H_4, and all intermediate results. The *message digest* unit outputs the final 160-bit message digest.

3.3. Application architecture

Figure 2 depicts the implementation of the PDEK application. The FPGA is connected externally to an ATA disk drive. The disk drive is used for storing the dictionary of passwords. The FPGA implementation expects the dictionary to be loaded into contiguous sectors on the disk a priori. An ATA controller is implemented on the disk and is used to interface with the disk. The controller streams passwords to a FIFO demultiplexer unit. The FIFO demultiplexer takes the password stream from the ATA controller and distributes the passwords to the input FIFOs of the four PDEK generation units.

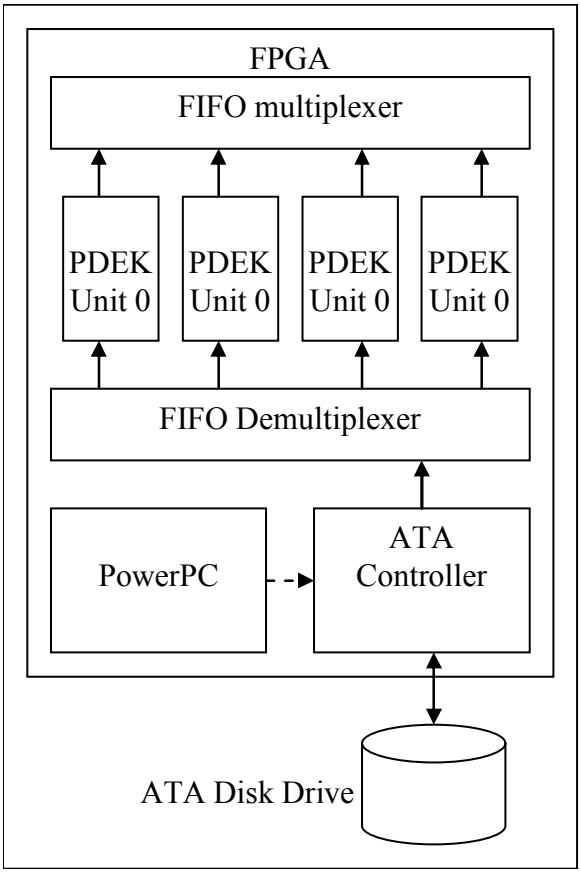

Figure 2: Application Block Diagram

Each PDEK unit removes a password from its input FIFO and generates the output PDEK. The generated PDEK is placed by the PDEK unit into the output FIFO multiplexer unit. For this implementation, the FIFO multiplexer simply discards the generated data. In an actual dictionary attack application, the generated PDEKs will be given to a software routine for attempting a decryption operation using the PDEK.

The PowerPC core controls the overall operation of the implementation. The software executing on the PowerPC issues instructions to the ATA controller for reading passwords from the disk and monitors the status if the FIFO input demultiplexer, the PDEK units, and the output FIFO multiplexer unit. The PowerPC software also measures performance statistics.

3.4. Latency analysis

Each pair of SHA-1 invocations (see lines 2 and 6 of the pseudo code in section 3) requires an additional overhead of three clock cycles for performing the required exclusive-or and concatenation operations. Furthermore, each of the c invocations of the HMAC-SHA-1 algorithm also requires an additional clock cycle for iteration control. Therefore, the total number of clock cycles required by c invocations of the HMAC-SHA-1 algorithm for a single PDEK is given by the following expression:

$$\lambda = 1 + c(1 + 3 + 2 \times 122) .$$

When c is 4,000, λ becomes 992,001 clock cycles (*i.e.*, it takes 992,001 clock cycles to generate a PDEK for a single password). Note that because acquiring passwords from the disk and placing passwords into the implementation FIFOs occurs simultaneously with the PDEK generation, those costs are not included in this analysis (*i.e.*, delays caused by these I/O operations is assumed to be insignificant because it is absorbed into the PDEK generation operations).

4. Experimental results

The design described in section 3 and Figure 2, in particular has been implemented in Verilog and synthesized using Xilinx's Embedded Development Kit (EDK) software. The target FPGA selected for synthesis is the Virtex 4 FX-100 with speed grade -11. The synthesized design can operate at a frequency of 162MHz. However, the initial design implementation was given a clock signal with a frequency of 150MHz (half the frequency of the embedded PowerPC processor core on the FPGA) in order to enable easy clock signal generation. A small fraction of the FPGA was used for this implementation – indicating that additional PDEK units could be incorporated into the design. This (and other) optimizations will be performed in the coming weeks.

A database of 10,000 English words at least 6 characters long is used for testing the throughput of the FPGA-based PDEK implementation. The 320,000 byte long word list is stored in 625 consecutive sectors on an IDE disk drive (each disk sector holds 512 bytes). The software executing on the FPGA PowerPC

core reads the data in the sectors into the FPGA and streams the 32-byte words into the input FIFOs of the PDEK modules. Because the disk controller operates at a peak transfer rate of 100 megabytes per second and the data is stored sequentially on the disk, the input FIFOs are always full of data until there are no more passwords to be processed (*i.e.*, disk and FIFO I/O is guaranteed not to introduce any delay except for the initial delay when the first disk sector is read in). Any initial I/O delay is amortized into the throughput computation for the 10,000 passwords.

The initial implementation consisting of four PDEK generation units operating in parallel takes a total of 19.589 seconds to generate PDEKs for the 10,000 passwords. Because the four PDEKs operate in parallel asynchronously from each other, it can be assumed that each PDEK unit processed approximately 2,500 passwords in the 19.589 seconds. Therefore, each unit takes 19.589s/2,500 or 0.0078356 seconds to produce a PDEK. Given a clock frequency of 150MHz, it takes $0.0078356 \times 150E^6$ or approximately 1,175,340 clock cycles to produce a PDEK result. This indicates that the implementation is taking nearly 16% more cycles than indicated by the theoretical analysis in section 3. However, some of the additional overhead can be attributed to the initial disk I/O latency. These performance figures also indicate that the initial implementation has introduced unintended additional pipeline delays that will need to be removed during a subsequent optimization effort.

5. Conclusions

In order to perform forensic analysis on data that has been encrypted using advanced password based encryption key techniques, forensic investigators will be required to perform dictionary based attacks in order to crack the passwords that have been used to encrypt the data. However, most modern PDEK techniques impose a significant computational burden on attackers by requiring a large number of calls to cryptographic encryption functions such as SHA-1. Software-based hash generation techniques are relatively slow and parallel implementations are traditionally employed in order to improve throughput.

This paper describes a hardware implementation using an FPGA platform for significantly improving the performance of the dictionary attack as compared to software implementations. The performance of the an example sequential (*i.e.*, non-parallel) software implementation written for comparative analysis is several order of magnitude slower than the FPGA-based implementation. Because of this, it is apparent that considerable optimization and parallelization effort will be required to make the software version competitive with the FPGA-version. Therefore, the software version has currently been set aside until the FPGA version is fully optimized.

The initial FPGA-based implementation is able to achieve a throughput of over 510 passwords per second using four PDEK generation units operating in parallel. Several optimizations to this initial implementation will be investigated in the near term. One optimization is to increase the parallelism by incorporating more parallel PDEK generation units. Another near-term optimization is to increase the clock frequency of the implementation.

Longer term optimizations requiring more effort will include the exploration the possibility of utilizing a cluster of FPGA platforms in parallel in order to further speedup the dictionary attack. Because the application is "embarrassingly" parallel, utilizing additional hardware should result in near-linear speedup. The FPGA platforms utilized in this project all have multiple gigabit-per-second capable network interfaces that can be employed to construct a high-speed distributed dictionary attack infrastructure. Another long-term optimization is to look for, and to eliminate, sources of latency that were inadvertently inserted into the implementation of the PDEK units.

6. References

1. NAI Labs, Network Associates, Inc., "Windows Data Protection," *Microsoft Developer Network*, October 2001, http://msdn2.microsoft.com/en-us/library/ms995355.aspx (accessed June 2007).

2. AccessData, "White Paper: Password Recovery with PRTK/DNA," March 2006, http://www.accessdata.com/media/en_US/print/papers/wp.PRTK-DNA_Password_Recovery.en_us.pdf (accessed June 2007).

3. F. Biancuzzi, "John the Ripper 1.7, by Solar Designer," *Security Focus*, February 22, 2006, http://www.securityfocus.com/columnists/388 (accessed June 2007).

4. P. Oechslin, "Making a Faster Crytanalytical Time-Memory Trade-Off," in *proceedings of the 23rd Annual International Cryptology Conference*, Santa Barbara, California, USA, August 17-21, 2003.

5. RSA Laboratories, "PKCS #5 v2.0: Password-Based Cryptography Standard," March 25, 1999, ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2/pkcs5v2-0.pdf (accessed June 2007).

6. Information Technology Laboratory – National Institute of Standards and Technology, "The Keyed-Hash Message Authentication Code (HMAC)," *FIPS PUB 198*, March 6, 2002, http://csrc.nist.gov/publications/fips/fips198/fips-198a.pdf (accessed June 2006).

7. Information Technology Laboratory – National Institute of Standards and Technology, Secure Hash Standard (SHS), *FIPS PUB 180–2*, August 2002, http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf (accessed June 2007).

8. A. P. Kakarountas, H. Michail, A. Milidonis, C. E. Goutis, and G. Theodoridis, "High-Speed FPGA Implementation of Secure Hash Algorithm for IPSec and VPN Applications," *The Journal of Supercomputing*, 37, pp. 179–195, 2006.

9. Y. K. Kang, D. W. Kim, T. W. Kwon, and J. R. Choi, "An Efficient Implementation of Hash Function Processor for IPSEC," in *Proceedings of the IEEE Asia-Pacific Conference on ASIC*, 2002.

10. R. Lien, T. Grembowski, and K. Gaj, "A 1 Gbit/s Partially Unrolled Architecture of Hash Functions SHA-1 and SHA-512," In *Proceedings of the RSA Conference*, Berlin Heidelberg, 2004.