Time-Memory Trade-Off Attack on FPGA Platforms: UNIX Password Cracking

Nele Mentens, Lejla Batina, Bart Preneel, and Ingrid Verbauwhede*

Katholieke Universiteit Leuven, ESAT/SCD-COSIC
Kasteelpark Arenberg 10
B-3001 Leuven-Heverlee, Belgium
{Nele.Mentens, Lejla.Batina, Bart.Preneel,
Ingrid.Verbauwhede}@esat.kuleuven.ac.be

Abstract. This paper presents a hardware architecture for UNIX password cracking using Hellman's time-memory trade-off; it is the first hardware design for a key search machine based on the rainbow variant proposed by Oechslin. The implementation target is the Berkeley BEE2 FPGA platform which can run at 400 million password calculations/second. Our design targets passwords of length 48 bits (out of 56). This means that with one BEE2 module the precomputation for one salt takes about 8 days, resulting in a storage of 56 Gigabyte. For the precomputation of all salts in one year we would need 92 BEE2 modules. Recovering an individual password requires a few minutes on a Virtex-4 FPGA.

Keywords: cryptanalysis, hash-functions, time-memory trade-off, exhaustive key search, rainbow table, FPGA implementation.

1 Introduction

Symmetric-key cryptography deals with algorithms that use a secret key to provide confidentiality, identification and data authentication. A basic problem in symmetric-key cryptology is the computation of preimages or inversion of one-way functions. For example, a brute-force attack on a block cipher in a known plaintext attack considers the mapping of the key to the ciphertext, which should be a one-way function. If no shortcut method is known, and the function has an n-bit result, there are two straightforward methods: first one can perform an exhaustive search over an average of 2^{n-1} values until the target is reached. A second solution is to precompute and store 2^n input and output pairs in a table (for a random function this will not result in different values – if the input space is large enough, the coupon collector's formula tells us that a space of about $n \cdot 2^n$ elements needs to be searched). If one then needs to invert a particular value, one just looks up the preimage in the table, so inverting requires only a single table lookup.

* Nele Mentens, Lejla Batina, Bart Preneel and Ingrid Verbauwhede are funded by FWO projects (G.0450.04, G.0141.03). This work is also supported by EU IST FP6 projects SCARD and ECRYPT and GOA Ambiorix 2005/11.

K. Bertels, J.M.P. Cardoso, and S. Vassiliadis (Eds.): ARC 2006, LNCS 3985, pp. 323–334, 2006.

The time-memory trade-off attack invented by Hellman in 1980 [7] proposes a solution that lies in between the two solutions. The precomputation time is still on the order of 2^n, but the memory complexity is $2^{2n/3}$ and the inversion of a single value requires only $2^{2n/3}$ function evaluations. In [6] Fiat and Naor propose a more general and rigorous variant at the cost of extra workload and/or memory. Kusuda and Matsumoto generalize the Hellman method in [8]; they derive stricter bounds on the success probability and give relationships between the memory complexity, processing complexity and success probability. Note that for cryptanalyzing stream ciphers more complex time/memory/data trade-offs are known – see for example Biryukov and Shamir [4].

Hellman's basic idea was improved in 1982 by Rivest who suggested to use distinguished points in order to reduce the number of memory accesses. This idea was elaborated independently by Borst *et al.* [5] and Stern [13]. The first FPGA design of this method was proposed by Quisquater *et al.* [12] for a 40-bit DES variant; they also presented cost estimations for the cryptanalysis of a full DES (with 56 bits). A detailed analysis for this platform was given in [15]. A more generic full cost analysis of the time-memory trade-off with and without distinguished points has been provided by Wiener in [16].

At Crypto 2003, Oechslin [10] suggested to use so-called rainbow tables for precomputations; this method combines the advantage of the distinguished point approach (reduced number of memory accesses) with the higher success probability and easier analysis of Hellman's original method. He has developed further details in [11].

In this paper we propose an FPGA platform for cryptanalysis of the UNIX password hashing scheme [9] using the rainbow table approach. The implementation target is an existing FPGA platform called BEE2 (Berkeley Emulation Engine 2) [1]. The paper is organized as follows. Section 2 provides the theoretical background and some definitions as well as specifics related to our case. In Sect. 3 the details of the proposed FPGA implementation are described together with future improvements. More future work is depicted in Sect. 4 and Sect. 5 concludes the paper.

2 Theoretical Background

In this section we give some definitions that are used in the remainder of the paper.

2.1 Time-Memory Trade-Off

Let $E : \{0,1\}^n \times \{0,1\}^k \longrightarrow \{0,1\}^n$ be a block cipher with block length n and key length k. We will consider DES with $n = 64$ and $k = 56$, or rather a variant of DES. The encryption is denoted as: $C = E_K(P)$ where C, K and P are respectively ciphertext, key and plaintext. For a fixed and known plaintext P, the mapping $E_K(P)$ is a one-way function from the key to the ciphertext. For a time-memory trade-off two functions are usually defined. The first one is $g : \{0,1\}^{64} \longrightarrow \{0,1\}^{56}$ that maps a ciphertext to a k-bit string, hence we can write:

$$g(C) = g(C_1, C_2, \ldots, C_{64}) = (X_1, X_2, \ldots, X_{56})\,. \tag{1}$$

This function is usually called a mask function or a reduction function. There are many possibilities to define this function; one often proposes to drop 8 bits and to permute the other 56 ones, which results in more than 2^{280} choices. Other options that are more suitable for hardware implementations include bit swaps, xor functions, etc. We discuss these options in more detail in Sect. 3.1. Second, we define a function $f : \{0,1\}^{56} \longrightarrow \{0,1\}^{56}$ that maps the key space to itself:

$$f(K) = g(E_K(P)) = g(C_1, C_2, \ldots, C_{64}) = g(C), \forall\, K \in \{0,1\}^{56}\,. \tag{2}$$

This construction originates from Hellman [7]; it was generalized by Kusuda and Matsumoto in [8]. By succession of ciphertexts with keys a chain can be constructed:

$$K_i \overset{E_{K_i}(P)}{\longrightarrow} C_i \overset{g(C_i)}{\longrightarrow} K_{i+1}\,,$$

which can be written as a chain of keys

$$K_i \overset{f}{\longrightarrow} K_{i+1} \overset{f}{\longrightarrow} K_{i+2}\,.$$

In the original algorithm of Hellman m chains of length t are created; one stores only the first and the last element of each chain in a table. Given a ciphertext C (with a known plaintext) one can try to find a key that was used to generate C in the following way. Chains (up to some fixed length t) are searched until a key that matches the last key of some chain is found. Using the first key, the chain can be reconstructed and the right key is the one just before C. The typical parameter sizes for a k-bit key are $t = m = 2^{k/3}$. If one uses $r = 2^{k/3}$ tables, the total precomputation time is 2^k evaluations of f and one needs to store $2^{2k/3}$ values of $2k$ bits. Recovering a single key requires $2^{2k/3}$ evaluations of f. The success probability of this method depends on the number of repeated elements in the chains; repetitions occur due to merging chains and due to chains that enter a loop. For the typical parameter sizes $t = m = r = 2^{k/3}$, with a precomputation complexity of 2^k, the success probability is around 0.55 [7].

The approach of distinguished points avoids a table lookup after every function computation, since an efficient implementation of a lookup in a large table would be too expensive. A distinguished point is a key that has a property that is easy to identify (for example the 20 most significant bits are zero); this means that one only needs to check after each iteration whether or not a value is a distinguished point. One creates chains starting and ending with a distinguished point: this also allows to reduce the storage per chain and to check for some merged chains (but throwing away such chains implies that one needs to increase the precomputation time). However, in the distinguished point variant, chains are of unequal length and will have a larger probability to merge (reducing the success probability of the attack).

The rainbow tables approach proposed by Oechslin [10] uses a different function g in every iteration. More precisely, rainbow chains have a fixed length t and use t different mask functions inside one chain: $g_1, \ldots, g_t$. In order to recover a key one first starts in the one but last column (1 application of g_t); next

one starts in the second but last column and one applies g_{t-1} and g_t. In the final iteration one applies g_1 through g_t; the total number of iterations is $\frac{t(t-1)}{2}$. This also allows to reduce the memory accesses, but at the same time it reduces the probability of merging chains; indeed, two chains will only merge if the two merging points are at the same position in a chain. Because of the reduced merge probability, rainbow tables can be much larger; typically only a few tables are needed [11]. The method has been implemented in software (a.o. for Windows passwords), but we are not aware of any hardware implementations. This article explores some options for hardware implementations of rainbow chains applied to the UNIX password system.

2.2 UNIX Password Hashing

Here we consider the application of the time-memory trade-off to the UNIX password system. In this case, 25 DES operations are performed where the ciphertext of one DES is used as the plaintext of the next DES. The plaintext of the first DES consists of all zeros and the key to all DES functions is the user password consisting of 8 ASCII characters. The ciphertext of the last DES block is the hash-value of the password. To increase the security of the UNIX password system, these 25 DES functions are modified based on a 12-bit salt; this salt defines an extra permutation in the expansion function in each round. The salt is a public value that is allocated to the user when she registers to the system; it is stored together with the hash value. The salt is often derived from the system clock. The black-box representation of this scheme is shown in Fig. 1. Assuming password characters consisting of capitals, small letters, numbers and two special characters " \" and "." every character contains only 6 bits of information which results in a key space of 48 bits. The password salting results in 2^{12} extra variations, hence the time-memory trade-off precomputation needs to be repeated for all salts: both the storage and the precomputation time increase with a factor 4096, but a single password can still be recovered with $2^{48 \cdot 2/3} = 2^{32}$ function evaluations. Of course one can also choose to mount the attack for a subset of salts.

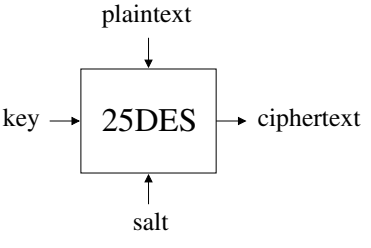

Fig. 1. Black-box of one UNIX password hashing

2.3 Bounds and Parameters

We now introduce some notation. Let t be the length of the chains, let m denote the number of chains in each table and r the number of tables. These parameters can be varied in order to tune the success rate as the time-memory trade-off is a probabilistic method. The schematics of one chain and the total structure are shown in Fig. 2 and Fig. 3. The bounds on the memory M (used to store the precomputation tables) and the time T (required to find the password starting from the hash) are as follows:

$$\begin{aligned} M &= m \cdot r \cdot m_0 \\ T &= t \cdot r \cdot t_0 \end{aligned}$$

Here, m_0 is the amount of memory required to store each chain *i.e.* its start and end point. In our case m_0 is 14 bytes. Likewise, t_0 is the time in which one password hash is generated.

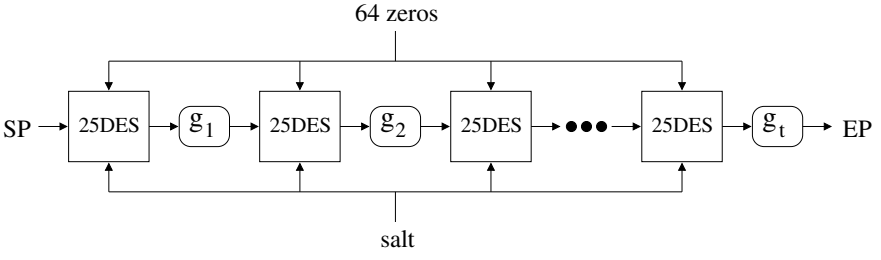

Fig. 2. Schematics of one rainbow chain. The inputs and outputs of one hash function are depicted in Fig. 1. SP = start point, EP = end point.

The success rate of a single rainbow table can be estimated as follows [10]:

$$P = 1 - \prod_{i=1}^{t} (1 - \frac{m_i}{N}) \,,$$

where

$$m_1 = m, \quad m_{n+1} = N(1 - e^{-\frac{m_n}{N}}) \,.$$

In Fig. 4 the success rate is shown as a function of the length of the chains t. It is obvious that the probability grows fast at the beginning with the length of the chains. After a length of around 102 400 ($\approx 2^{16.64}$) the probability is almost stagnating.

By taking the direct approach from the original paper of Hellman we derived the following lower bound. By approximating $mt^2 \approx N$ the lower bound can be estimated to be 0.75, which is similar to the result of Standaert *et al.* [15]:

$$P \geq \frac{mt}{N} \Big[1 - \frac{mt^2}{4N} + \frac{(mt^2)^2}{18N} - \frac{(mt^2)^3}{96N} + \frac{(mt^2)^4}{600N} - \cdots \Big] \tag{3}$$

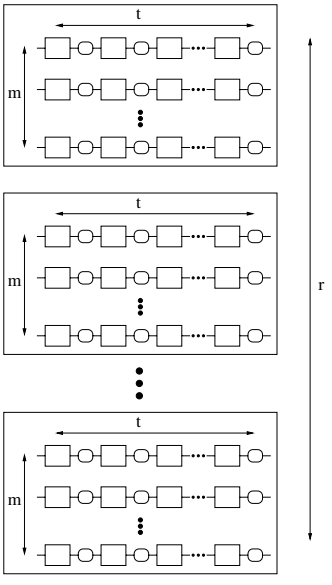

Fig. 3. Schematics and parameters of the complete structure

The success rate for multiple tables can be calculated from

$$P \geq 1 - (1 - P_{one\,table})^r \,. \tag{4}$$

Here we consider only one rainbow table, which can be justified by the result of Oechslin. In his work the best cryptanalysis results were achieved using only five tables.

3 Hardware Implementation Options and Results

We now elaborate on the implementation of the precomputation in hardware. We describe the FPGA design and give performance estimates.

3.1 Our FPGA Solution

The first crucial choice is related to the mask functions. The mask function is actually a reduction function that maps a ciphertext to a key. There exist various options among which we mention:

- permutations *i.e.* S-boxes
- xor functions
- bit swaps

As we are interested in hardware implementations, it is important to choose mask functions with a low hardware complexity. From this point of view, all

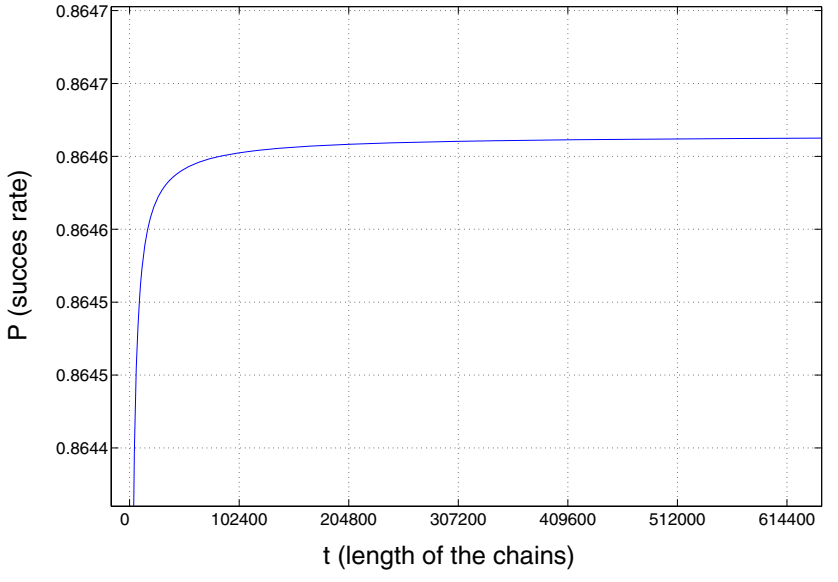

Fig. 4. Success rate as a function of the length of the chains

three suggested options are suitable. However, for rainbow tables a chain contains many different mask functions which implies that the overhead in control logic should also be minimized. With respect to this, xor-ing with a register containing a variable value is the best solution. Moreover, in our case permutations may not even offer enough choices for different masks. Since the complexity of our key space is 2^{48} we chose to throw away the last 16 bits and to xor with a 48-bit counter. In this way, we can use just one generic mask function which is varied by different states of the counter resulting in a total of 2^{48} different mask functions. Finally, the 48-bit output of the xor function passes through some logic gates to obtain a 56-bit result which is a valid 56-bit key (containing only capitals, small letters, numbers and the characters " \" and ".").

Fig. 5 depicts the architecture of our design. To construct a chain, an alternating sequence of block cipher computations and mask functions is applied. This is done using a feedback loop.

The generation of start points is implemented in hardware in order to contribute to a more efficient precomputation. More precisely, loading start points of chains from outside of the FPGA would create an overhead in communication time. Namely, because of pipelining, the design has to deal with many start points at the same time. For this reason, we implemented a counter to generate the start points. The value of the counter in the mask functions, padded with 8 zeros, can be re-used for this purpose.

Next a buffer design needs to be developed to take into account the variable output rate of the rainbow algorithm. The start point-end point pairs are stored in a hash table with the end point as the index. After sorting the table entries,

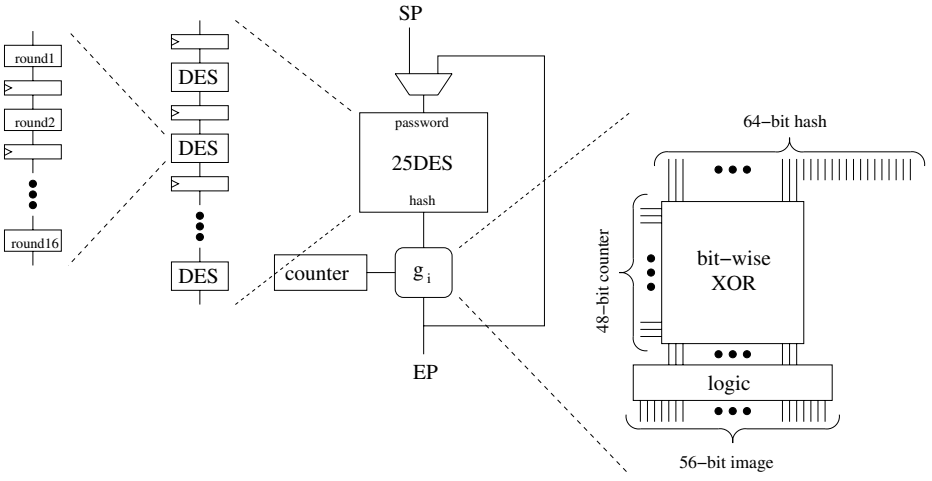

Fig. 5. Pipelined architecture for performing the rainbow chains

the on-line part has to be performed. It looks up the values output by the FPGA until the targeted key is found. Recovering an individual password in a rainbow chain takes $\frac{t(t-1)}{2}$ function evaluations. This part can be done in software or, to make it faster, on an FPGA. Using a XC4VLX200 Virtex-4, a non-pipelined version of the Unix password system can find the key in less than an hour. A pipelined key search on this FPGA can be done in a few minutes.

There are two cases when finding an end point does not lead to the correct key; these are usually referred to as false alarms. First, the key may be a part of a chain with the same end point but which is not in the table. The second false alarm situation occurs when a key is in a chain which merges with another chain in the table. For rainbow chains, the merging will occur only if the collision happens at the same position in two chains. For chains of length t, the probability that a collision is a merge is only $\frac{1}{t}$. As noted in [10], it is possible to generate tables without any merging. However, this solves only some false alarm situations and it remains a problem to create tables that cover the key space as much as possible.

3.2 Hardware Precomputation Platform and Results

We chose an existing platform to perform the precomputation part: the Berkeley Emulation Engine 2 (BEE2) [1]. One BEE2 module consists of 5 Xilinx Virtex-2Pro70 FPGAs of which 4 can be used to implement digital circuits and one to take care of global routing and control logic. The floorplan of one module is depicted in Fig. 6. The Virtex-2Pro70 is a high performance FPGA which comes at a cost of approximately US$ 1500. The BEE2 platform is designed for high-speed applications with a communication bandwidth up to 360 Gbit/s. Every module contains a 20 GB DDR-RAM and a 10 Gbit/s ethernet connector.

However, our application only requires a 14-byte value to be written in the RAM after every 2^{16} hash operations, which comes down to a write speed of 85 449 bytes/s for one BEE2 module. This is achievable by a hard disk supplemented with 500 MB or 1 GB of RAM. Furthermore, for recovering one key, we need approximately 2^{32} 7-byte table lookups in a 56 GB memory. Assuming it takes 2 minutes to perform the on-line computations, this comes down to 36 million read operations per second at a bandwidth to the memory of 250 MB/s. This means that for the on-line part it is also not required to have a bandwidth of 360 Gbit/s nor a 20 GB RAM. Hence, a dedicated design with a slower memory access rate could reduce the full cost.

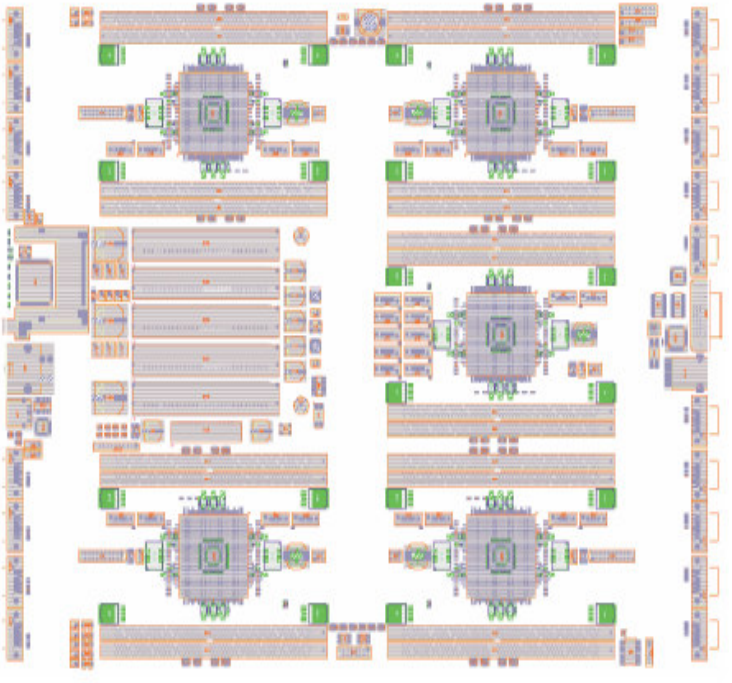

Fig. 6. Floorplan of one BEE2 module

Synthesis results show that one precomputation unit uses almost 200% of the slices of a Virtex-2Pro70 (33 088 slices). That is why we use one BEE2 module to implement two precomputation units. Each pipelined architecture in Fig. 5 can compute 200 million password hashes per second at a frequency of 200 MHz. Targeting 48-bit passwords using one BEE2 module, this would mean a precomputation time of 8 days. The upper bound on the storage for one salt is $2^{32} \times$ 14 Bytes, resulting in 56 GB of memory. To make the precomputation for all salts in one year, we need 92 BEE2 modules. Figures 7 and 8 depict the precomputation time as a function of the number of BEE2 modules used in parallel.

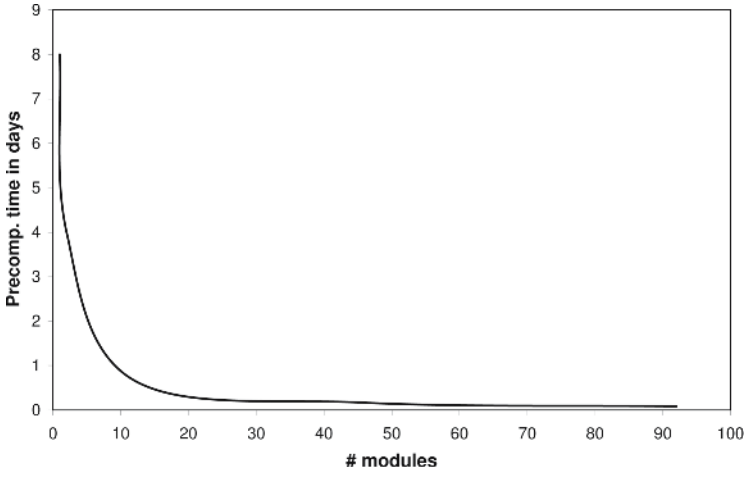

Fig. 7. Precomputation time for one salt as a function of the number of BEE2 modules

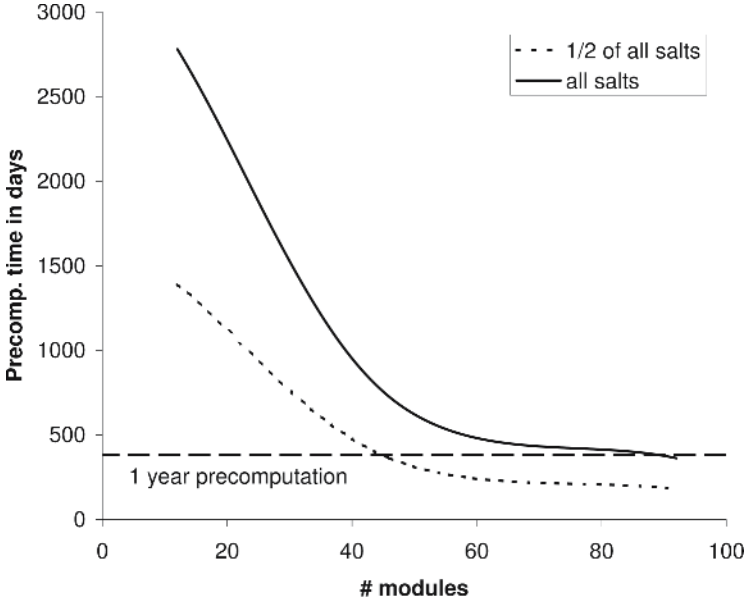

Fig. 8. Precomputation time for half of the salts and for all salts as a function of the number of BEE2 modules

Table 1 compares our results with other hardware as well as software solutions. The only known hardware solution is [12] which attacked one 40-bit DES while our target was 25 56-bit DES blocks. The other are software options dedicated to cracking 56-bit DES.

Table 1. Comparison of implementation results for symmetric key cryptanalysis

	platform	algorithm	speed (enc/s)
[2]	64-bit Alpha computer	56-bit DES	2 M
[12]	Virtex1000	40-bit DES	66 M
[10]	P4, 1.5 GHz, 500 MB RAM	56-bit DES	0.7 M
this work	BEE2	25 x 56-bit modified DES	400 M

4 Future Work

The results described in this paper can be further optimized by considering the work of Biryukov *et al.* [3], in which time-memory-data trade-off attacks show an improvement of a factor 2 to 3.

Another standard for computing UNIX password hashes is based on the MD5 algorithm [14]. The feasibility of attacking these kinds of UNIX password systems should be investigated.

5 Conclusions

In this paper we presented an FPGA architecture for cracking UNIX passwords using the rainbow tables approach from Oechslin. The attack targets passwords consisting of capitals, small letters, numbers and a few special characters, *i.e.* 48-bit passwords. The implementation platform consists of BEE2 modules developed at UC Berkeley. We give the implementation results for one BEE2 module precomputing the rainbow tables for one salt. Furthermore, we estimate the number of modules needed for the precomputation of all salts in one year.

References

1. University of California Berkeley Wireless Research Center. Bee home page. `http://bwrc.eecs.berkeley.edu/Research/BEE/`.
2. E. Biham. A fast new DES implementation in software. In E. Biham, editor, *Proceedings of 4th International Workshop on Fast Software Encryption Workshop (FSE)*, number 1267 in Lecture Notes in Computer Science. Springer-Verlag, 1997.
3. A. Biryukov, S. Mukhopadhyay, and P. Sarkar. Improved time-memory trade-offs with multiple data. In B. Preneel and S. Tavares, editors, *Proceedings of the 12th Annual Workshop on Selected Areas in Cryptography*, Lecture Notes in Computer Science, page 19 pages. Springer, 2005.
4. A. Biryukov and A. Shamir. Cryptanalytic time/memory/data tradeoffs for stream ciphers. In Tatsuaki Okamoto, editor, *Advances in Cryptology: Proceedings of ASIACRYPT*, volume 1976 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2000.
5. J. Borst, B. Preneel, and J. Vandewalle. On the memory trade-off between exhaustive key-search and table precomputation. In *Proceedings of the 19th Symposium on Information Theory in the Benelux*, pages 111–118. Werkgemeenschap voor Informatie- en Communicatietheorie, Enschede, The Netherlands, 1998.

6. A. Fiat and M. Naor. Rigorous time/space tradeoffs for inverting functions. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pages 534–541, 1991.
7. M. Hellman. A cryptanalytic time-memory trade-off. *IEEE Transactions on Information Theory*, 26:401–406, 1980.
8. K. Kusuda and T. Matsumoto. Optimization of time-memory trade-off cryptanalysis and its application to DES, FEAL-32 and Skipjack. *IEICE Transcations on Fundamentals of Electronics, Communications and Computer Science*, E-79A:35–48, 1996.
9. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.
10. P. Oechslin. Making a faster cryptanalytic time-memory trade-off. In D. Boneh, editor, *Advances in Cryptology: Proceedings of CRYPTO*, number 2729 in Lecture Notes in Computer Science, pages 617–630. Springer-Verlag, 2003.
11. P. Oechslin. Les compromis temps-mémoire et leur utilisation pour casser les mots de passe Windows. In *Symposium sur la Sécurité des Technologies de l'Information et de la Communication SSTIC, Rennes*, June 2004.
12. J.-J. Quisquater, F.-X. Standaert, G. Rouvroy, and J.D. Legat. A cryptanalytic time-memory trade-off: First FPGA implementation. In *Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications (FPL)*, volume 2438 of *Lecture Notes in Computer Science*, pages 780–789. Springer-Verlag, 2002.
13. J.-J. Quisquater and J. Stern. Time-memory tradeoff revisited. *Unpublished*, 1998.
14. R. Rivest. The MD5 Message-Digest Algorithm. `http://www.ietf.org/rfc/rfc1321.txt` , 1992.
15. F.-X. Standaert, G. Rouvoy, J.-J Quisquater, and J.-D. Legat. A time-memory trade-off using distinguished points: New analysis and FPGA results. In B. S. Kaliski Jr., Ç. K. Koç, and C. Paar, editors, *Proceedings of 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, number 2535 in Lecture Notes in Computer Science, page 593609. Springer-Verlag, 2002.
16. M. J. Wiener. The full cost of cryptanalytic attacks. *Journal of Cryptology*, 17(2):105–124, 2004.