Cracking Unix Passwords using FPGA Platforms

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 the rainbow variant proposed by Oechslin. We estimate that an FPGA implementation of the function can run at 17.5 million password tests/second on a Virtex-4. Our design targets passwords of length 48 bits (out of 56). This means that with 16 FPGAs the precomputation for one salt takes about 11 days, resulting in a storage of 56 Gigabyte. Recovering an individual password requires a few minutes.
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 only a secret key to provide confidentiality, identification and data authentication. A fundamental 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 plaintext, 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 on 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.

The time-memory trade-off attack invented by Hellman in 1980 [5] 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 [4] 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 [6]; 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 [2].

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.* [3] and Stern [11]. The first FPGA design of this method was proposed by Quisquater *et al.* [10] 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 [12]. A more generic full-cost analysis of the time-memory trade-off with and without distinguished points has been provided by Wiener in [13].

At Crypto 2003, Oechslin [8] 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 [9].

In this paper we propose an FPGA platform for cryptanalysis of the UNIX passwords hashing scheme [7]. We use the rainbow table approach and we give some first estimations and results. This 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 new FPGA implementation are described together with future improvements. Finally, Sect. 4 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 an extension of DES with $n = 64$ and $k = 56$. The encryption can be written 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 key-length string, so 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 Section 3.1 in more detail. Second, we define a function $f : \{0,1\}^{56} \longrightarrow \{0,1\}^{56}$ that maps the key-space to itself:

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

This construction originates from Hellman [5]; it was generalized by Kusuda and Matsumoto in [6]. By succession of ciphertexts with keys a chain can be constructed:

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

which can be written as a chain of keys

$$K_i \xrightarrow{g} K_{i+1} \xrightarrow{g} K_{i+2}.$$

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 that are stored (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 $g(C)$. The typical parameter sizes for a k-bit key are $t = m = 2^{k/3}$. If one uses $2^{k/3}$ 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 approach of distinguished points avoids that one needs to look up a value in a table after every function computation, since this 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 should be zero); this means that one only needs to check after each iteration whether or not a value is a distinguished point or not. 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 table approach proposed by Oechslin [8] 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 $t(t-1)/2$. 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 [9]. 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 of the round function. 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, and 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^{32} function evaluations. Of course one can also choose to mount the attack for a subset of salts.

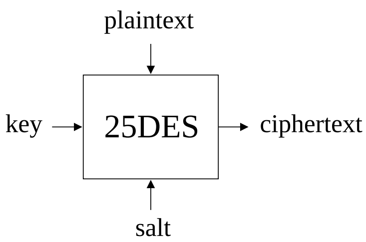

Fig. 1. Black-box of one UNIX password hashing.

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 starting and end point. In our case m_0 is 14 bytes. Likewise, t_0 is the time in which one password hash is generated.

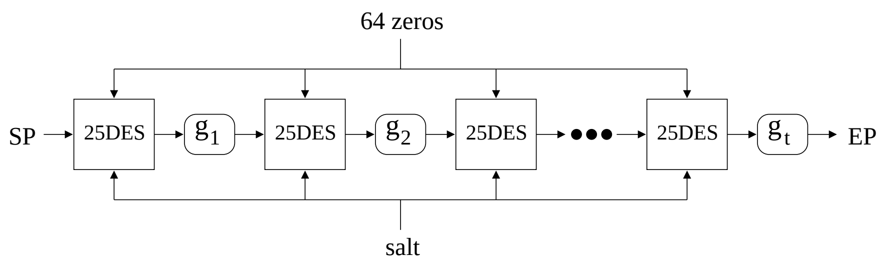

Fig. 2. Schematics of one rainbow chain. The inputs and outputs of one hash function are depicted in Fig. 1. SP = start point, EP = end point.

3 Hardware implementation options and results

Here we elaborate on the precomputation that has been implemented in hardware. We describe the FPGA design and give performance results.

Mask functions. 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 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.

For our implementation we chose to xor with a 56-bit counter. The last 8 bits are thrown away before xor-ing. 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^{56} different mask functions.

Generation of start points (SP). This task 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. For this reason, we implemented a counter to generate the start points. The counter we apply to construct the mask functions can be re-used for this purpose.

Our FPGA solution. Fig. 4 depicts the architecture of our design. To construct a chain, an alternating sequence of hash algorithms and mask functions is applied. This is done using a feedback loop.

We synthesized our design for a XC4VLX200 FPGA. The results are summarized and compared with other relevant work in Table 1.

Table 1. Comparison of implementation results for symmetric key cryptanalysis

	platform	algorithm	speed (enc/s)
[1]	software	56-bit DES	2 M
[10]	Virtex1000	40-bit DES	66 M
[8]	software	56-bit DES	0.7 M
this work	Virtex-4	25 x 56-bit modified DES	0.7 M

In this table we compared our results with other hardware as well as software solutions. The only known hardware solution is [10] which has a better performance figure due to pipelining and a much shorter algorithm: indeed, they attacked one 40-bit DES while our target was 25 56-bit DES blocks. The other are software options dedicated to cracking symmetric-key algorithms.

The next step in the development of this architecture is introducing pipelining which is expected to speed up the encryption time with a factor 25. Next a buffer design needs to be developed to take into account the variable output rate of the rainbow algorithm. After sorting the table entries, the on-line part has to be performed. It looks up the values output by the FPGA until the targeted key is found. This part can be done in software or, to make it faster, on an FPGA.

4 Conclusions

In this paper we presented an FPGA architecture for cracking UNIX passwords. The first indicative performance results are given and compared to other known work. The next development steps and further improvements are ongoing and future work. We plan to finalize a complete design by the time of the workshop (3rd week of February 2005) and hope to demonstrate the results (live?) in Paris.

References

1. 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.
2. A. Biryukov and A. Shamir. Cryptanalytic time/memory/data tradeoffs for stream ciphers. In Tatsuaki Okamoto, editor, *Advances in Cryptology - ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2000.
3. 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. Werkgemeeschap voor Informatie-en-Communicatietheorie, Enschede, The Netherlands, 1998.
4. 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.
5. M. Hellman. A cryptanalytic time-memory trade-off. *IEEE Transcactions on Information Theory*, 26:401–406, 1980.
6. K. Kusuda and T. Matsumoto. Optimization of time-memory trade-off cryptanalysis and its application to DES, FEAL-32 and Skip-jack. *IEICE Transcations on Fundamentals of Electronics, Communications and Computer Science*, E-79A:35–48, 1996.
7. A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1997.
8. P. Oechslin. Making a faster cryptanalytic time-memory trade-off. In D. Boneh, editor, *Advances in Cryptology: Proceedings of CRYPTO'03*, number 2729 in Lecture Notes in Computer Science, pages 617–630. Springer-Verlag, 1986.
9. 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.
10. 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.
11. J.-J. Quisquater and J. Stern. Time-memory tradeoff revisited. *Unpublished*, 1998.
12. 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.
13. M. J. Wiener. The full cost of cryptanalytic attacks. *J. Cryptology*, 17(2):105–124, 2004.

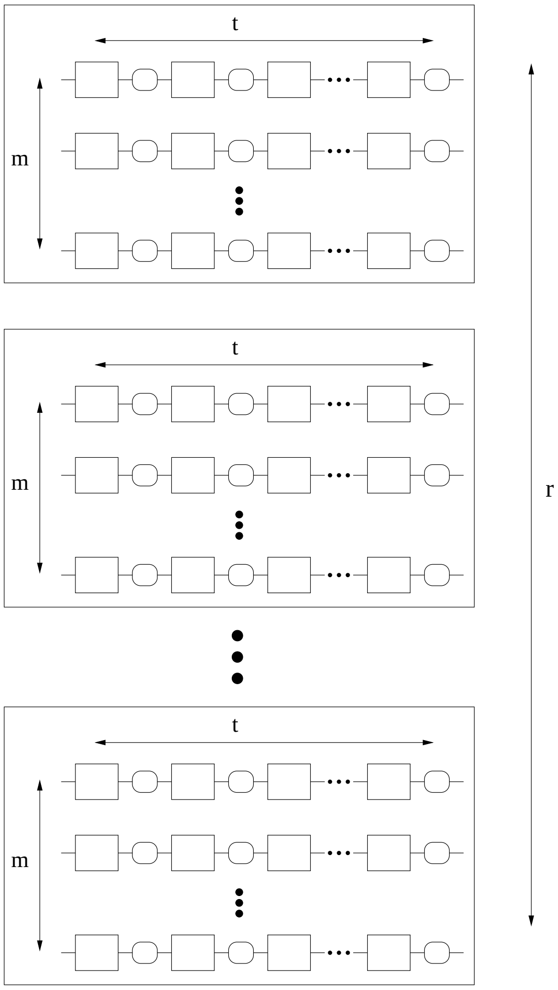

Fig. 3. Schematics and parameters of the complete structure.

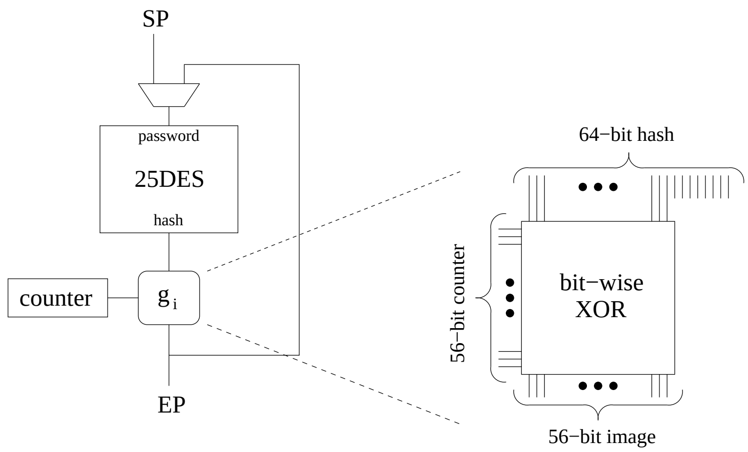

Fig. 4. Architecture for performing the rainbow chains.