J Comput Virol (2008) 4:73–81
DOI 10.1007/s11416-007-0064-y

SSTIC 2007 BEST ACADEMIC PAPERS

Advances in password cracking

Simon Marechal

Received: 5 January 2007 / Revised: 15 July 2007 / Accepted: 4 September 2007 / Published online: 11 October 2007
© Springer-Verlag France 2007

Abstract This paper surveys various techniques that have been used in public or privates tools in order to enhance the password cracking process. After a brief overview of this process, it addresses the issues of algorithmic and implementation optimisations, the use of special purpose hardware and the use of the Markov chains tool. Experimental results are then shown, comparing several implementations.

1 The password cracking process

Password-based authentication schemes work by comparing user's supplied passwords with stored secrets. As system administrators or hackers gaining equivalent privileges are allowed to access these secrets, passwords are usually not stored in plaintext. Most of the time, a cryptographic hash function is used. Cracking passwords is the process of getting the plaintext passwords from the stored secrets, or at least an equivalent one, which collides with respect to the hash function used, with the user's one.

Password cracking has seen few improvements compared to other IT security fields. Moreover, it has been shown that the knowledge of the password hash is sufficient for several authentication protocols. Malicious users now target the plaintext passwords, through phishing or keylogging. In some other cases, malware can directly look for files containing hash values of passwords. The password cracking field is now the home of security professionals and benchmark fanatics. They are however several known techniques that could be used to dramatically enhance the password cracking process.

Most password hashing schemes work by entering the plaintext password with a value that should be different for all users on the system (the *salt*) into a one way function. The result is stored with the salt. In order to crack passwords, one must:

- select candidate passwords that are likely to be chosen by users;
- get the salt corresponding to the passwords one's wants to crack;
- hash the candidate password with the salt and match it with the stored password hash.

In order to maximize cracking ability and efficiency, the cracking process should be sped up as much as possible, and the candidate password selection should be as smart as possible.

This paper presents an up-to-date survey of the most efficient techniques dedicated to password cracking. It is organized as follows. Section 2 discusses the main implementation issues with respect to password cracking techniques. They are related to the last known implementation improvements so far. Section 3 presents a few existing hardware architectures dedicated to password cracking. Section 4 introduces Markov chains as a powerful tool for improving distributed and rainbow table cracking. Section 5 presents some detailed results of experiments we have conducted by means of the techniques presented in this paper. Finally, we conclude in Sect. 6 by evoking the impact of these techniques in the context of malware attacks.

S. Marechal (✉)
Ecole Supérieure et d'Application des Transmissions,
avenue de la Touraudais, BP 18 35998 Rennes, France
e-mail: simon.marechal@esat.terre.defense.gouv.fr;
simon@banquise.net

2 Implementation optimisations

2.1 Reducing the instruction count

Reducing the quantity of instructions to be executed during the password hashing process is an obvious way to gain speed. Here are two shortcuts to achieve that goal, one of them being still widely used.

2.1.1 Implementing a reduced hash function

A typical cryptographic hash function works by:

- initializing some values and allocate memory to an internal buffer;
- copying the text to hash to an internal buffer, and padding it so that its length becomes a multiple of the block size (64 bytes for MD4, MD5, SHA) of the function;
- running the function "body" for every block but the last one;
- inserting a size dependent value to the last block;
- running the function "body" again on that last block;
- formatting the result and outputting it.

The first observation to be made is that the password length is very likely to be much less than the block size. That means that only the last block is to be hashed. The second observation is that the final step is very easy to reverse. For example, if an endianity conversion is necessary (from little-endian to big-endian or conversely), it could be performed on the hash one's tries to crack at the beginning of the cracking session, instead of doing it after each invocation of the hash function. Moreover, the internal buffer can be used again. The hashing process thus becomes:

- initializing some values;
- copying the hash value to an internal buffer. No padding is necessary as the internal buffer is already properly formatted;
- inserting a size dependent value in the last block;
- running the function "body" on the last block;

Some implementation generate the candidate password directly into the internal buffer, saving a memory copy instruction. It is worth mentioning that many popular password cracking tools do not even feature this simple optimisation. A notorious example is RainbowCrack [1], the most popular rainbow table cracker. It has become now famous due to its large file output and its use of the general purpose hashing functions of OpenSSL.

2.1.2 "Reversing" the hash function

The last rounds of the MD5 function "body" are described in Code sample 1. The "STEP" macro is described in Code sample 2. The resulting hash of the MD5 function is the concatenation of a, b, c, d values. When trying to crack a specific MD5 password, it is easy to infer the values of a, b, c and d at the end of step 0x3f. An important observation is that if the plaintext password is known, the whole hashing process can be reversed, and the values of a, b, c, d guessed at various stages of the function body. While it is not ground-breaking, this observation however leads to a smart optimisation.

Code sample 1 Last rounds of the MD5 body function

```
/* round 4 */
    STEP(I, a, b, c, d,  0, 0xf4292244, 6 ) /* step 0x30 */
    STEP(I, d, a, b, c,  7, 0x432aff97, 10) /* step 0x31 */
    STEP(I, c, d, a, b, 14, 0xab9423a7, 15) /* step 0x32 */
    STEP(I, b, c, d, a,  5, 0xfc93a039, 21) /* step 0x33 */
    STEP(I, a, b, c, d, 12, 0x655b59c3, 6 ) /* step 0x34 */
    STEP(I, d, a, b, c,  3, 0x8f0ccc92, 10) /* step 0x35 */
    STEP(I, c, d, a, b, 10, 0xffeff47d, 15) /* step 0x36 */
    STEP(I, b, c, d, a,  1, 0x85845dd1, 21) /* step 0x37 */
    STEP(I, a, b, c, d,  8, 0x6fa87e4f, 6 ) /* step 0x38 */
    STEP(I, d, a, b, c, 15, 0xfe2ce6e0, 10) /* step 0x39 */
    STEP(I, c, d, a, b,  6, 0xa3014314, 15) /* step 0x3a */
    STEP(I, b, c, d, a, 13, 0x4e0811a1, 21) /* step 0x3b */
    STEP(I, a, b, c, d,  4, 0xf7537e82, 6 ) /* step 0x3c */
    STEP(I, d, a, b, c, 11, 0xbd3af235, 10) /* step 0x3d */
    STEP(I, c, d, a, b,  2, 0x2ad7d2bb, 15) /* step 0x3e */
    STEP(I, b, c, d, a,  9, 0xeb86d391, 21) /* step 0x3f */
    a += 0x67452301;
    b += 0xe\-fc\-dab89;
    c += 0x98bad\-cfe;
    d += 0x10325476;
```

Code sample 2 The STEP macro for MD5

```
#define STEP(f, a, b, c, d, block, constant, rotation) \
    a += f(b, c, d) + buffer[block] + constant;
    a = ROTATE(a, rotation); \
    a += b;
```

When trying candidate passwords, we will suppose that the real password is partially known. For example, let us suppose that the first four bytes are unknown while the others are known. That way, it is easy to see that the values of a, b, c, d could be computed up to the end of step 0x30. Moreover, the values of b, c, d could be computed up to the end of step 0x2f, c, d for step 0x2e and finally d for step 0x2d.[1] Once this computation is achieved, the first four bytes could be brute-forced by starting the computation of the corresponding hash, stopping at step 0x2d and comparing the value of d instead of calculating until step 0x3f and comparing a, b, c, d. This increases the probability of finding a false positive from

[1] It is actually possible to go up to step 0x2b under the hypothesis that only a single byte is unknown, but in this case, the rate of false positives increases significantly.

2^{-128} [2] to 2^{-32}, which is still bad enough to be of practical interest.

Computing only 47 steps instead of 64 results in a speedup of 36%. However, this only works if candidate passwords are tested in an order that keeps the last bytes unmodified, making this technique especially suited for "stupid" brute force software that test passwords sequentially. This optimization, while known, is not implemented in any popular password cracker.

2.2 Writing optimized assembly code

The topic of writing optimized programs is a broad one. There are however specific tricks related to password cracking, most of them being related to the fact that password cracking techniques are easy to parallelize.

2.2.1 Filling the pipeline

Modern processors feature an instruction decoding pipeline. Instructions go through this pipeline and are effectively executed at its exit. Several instructions could be loaded in the pipeline at the same time. In order to maximize throughput, the pipeline must be kept as full as possible. However, as instructions are only effective at the exit of the pipeline, no instructions that use an input value that is the result of an instruction located in the pipeline could be loaded. The consequence is that several clock cycles are wasted until the value is computed and the next instruction could be loaded. This is called "pipeline starvation", caused by "instruction dependencies".

Here is sample MMX program:

```
pxor %{xmm}1, %{xmm}2
pand %{xmm}3, %{xmm}2
pxor %{xmm}4, %{xmm}2
```

This program performs the following operation:
`xmm2 = ((xmm2 ^ xmm1) & xmm3) ^ xmm4`
However, the `xmm2` register is the source and destination of all instructions in this short example. This means that the pipeline will be completely starved during the execution of this program. If the pipeline is n stages long, and the instructions have a single clock cycle latency, the previous program should take $3n$ cycles to execute. As four 32 bits values are calculated at once, it would take $3n/4$ cycles per result. In order to improve the attack efficiency, it could be possible to work on twice as much data by almost the same time by doing:

```
pxor %{xmm}1, %{xmm}2
pxor %{xmm}5, %{xmm}6
pand %{xmm}3, %{xmm}2
pand %{xmm}7, %{xmm}6
pxor %{xmm}4, %{xmm}2
pxor %{xmm}8, %{xmm}6
```

[2] Which would be as good as the real password!

This program should take $3n + 1$ cycles to execute, but should compute twice as much data ($(3n + 1)/8$ cycles per result). Working on as many password as possible at the same time is an excellent way to boost performance, because it is particularly well-suited to password cracking and easy to implement.

2.2.2 SIMD instructions

SIMD instructions are instructions that perform a single action on multiple data at once. On x86 processors, the MMX (64 bits) and SSE (128 bits) instruction set are SIMD instruction sets. Hashing functions such as MD4, MD5 and SHA1 are straightforwardly converted for the MMX (two passwords at once) or SSE (four passwords at once) instruction set. Currently, the John the Ripper tool [6,7] implements such optimizations, distributed as third party contributions.

3 Special purpose architectures

3.1 FPGAs

Field-programmable gate array (FPGAs) are devices that contain fully programmable logic. Designing FPGA cores is very different from standard CPU programming and a lot harder to debug. Development cost for a FPGA solution is much more important than for a software solution. It is however supposedly much more cost effective, as the FPGA chips will deliver much more performance than a general purpose CPU at identical costs. As far as password cracking is concerned, two projects are worth mentioning.

- The OpenCipher project, by David Hulton. The website states that "this sourceforge project is dedicated to exploring the uses of ASICs, FPGAs and other forms of programmable hardware with modern cryptography" [2].
- The Copacobana project [3], which claims impressive performances for a very low price: average DES key recovery in 7.2 days for 10,000$.

3.2 The CELL processor

The CELL processor, developed by Sony, Toshiba and IBM, is the core processor of the PlayStation 3 console. It features a PPC64 called the *Power Processing Element* (PPE) and seven *Synergistic Processing Elements* or SPEs, running at 3.2 GHz. The PPC64 core is a standard stripped-down

PowerPC. The SPUs are specialized processors featuring only SIMD instructions not unlike the MMX instruction set from Intel. The instruction set is however much larger and each SPU embeds 128×128-bits general purpose registers. Each SPU has a 256 kb "*local store*", a very fast memory located on the processor die. SPUs have however no direct access to the "*global store*" (the RAM system). The programmer must handle data exchange between the stores. As a comparison with general purpose CPUs, this would mean that the programmer should handle the CPU cache.

This architectural choice results in a system that is hard to master, and hard to optimize for arbitrary algorithms. It is however fairly easy to reach the theoretical performance peak on several applications. We managed to reach 180M passwords/s for MD4 at first try, with no real optimization effort, and without the MD4 "reversing" method. A quite good implementation runs at 11.5M passwords/s on an AMD64 3500+ (2.2 GHz).

4 Markov chains

Markov chains are mathematical tools that have been applied to password cracking by Arvind Narayanan and Vitaly Shmatikov [5]. They used it them improve rainbow table cracking, but it also has an important application for distributed password cracking.

4.1 Description

This method works by assuming that people select passwords whose character obey a hidden Markov model. That means that the probability that the nth character of the password is x, is a function of the previous characters. The probability of appearance of a particular password is the product of the probabilities of appearance of all the characters composing it. Without loss of generality, only first order Markov chains are studied in this paper but the probability of appearance of a given character is only a function of the previous character:

$$P(rabbit) = P(r) * P(a|r) * P(b|a) * P(b|b) * P(i|b) * P(t|i).$$

Here $P(x|y)$ is the probability of appearance of x after y has occured. The value of every $P(x|y)$ is approximated by performing frequency analysis on a large enough dictionary. A slight modification of the previous formula is useful when the actual implementation takes place. Let's state that $P'(a|b)$, the "corrected Markov probability" is equal to round$(-Klog(P(a|b)))$. In this way, multiplying floating numbers can be equivalent to adding integers:

$$P'(rabbit) = P'(r) + P'(a|r) + P'(b|a) + P'(b|b) + P'(i|b) * P'(t|i).$$

Let us call $P'(password)$ the Markov strength of the word "password".

4.2 Properties

The most interesting property of this statement lies in the fact that it is fairly easy to compute $nbp(previous, position, level, max)$, which is the number of passwords where the $position$th character is $previous$, the sum of the P' up to the $(position - 1)$-th character is $level$, and whose Markov strength is less than max.

A special value is $nbp('', 0, 0, max)$, the total number of passwords whose Markov strength is less than max. When writing the actual implementation, a size limit to the passwords is to be selected in order to compute a table nbp_{max} that fits in memory.

Sample C code 3 could be used to compute this value. The arrays "proba1" and "proba2" are listing the corrected Markov probability of appearance of a single character at the beginning of the password ("proba1") or after another character ("proba2"). It is worth mentioning that all admissible values are computed recursively when calculating $nbp('', 0, 0, max)$.

Code sample 3 Sample C code for the calculation of *nbp*

```
unsigned long long nb_parts(unsigned char previous,
   unsigned int position, unsigned int level,
   unsigned int max, unsigned int max_length)
{
 int i;
 unsigned long long out=1;

 if(level>max)
  return 0;

 if(position==max_length)
 {
  nbparts[previous + position*256 +
      level*256*max_length] = 1;
  return 1;
 }

 if(nbparts[previous + (position)*256 +
     level*256*max_length] != 0)
  return nbparts[previous + (position)*256 +
      level*256*max_length];

 for(i=1;i<256;i++)
  if(position==0)
   out += nb_parts(i, position+1, proba1[i],
       max, max_length);
  else
   out += nb_parts(i, position+1, level +
       proba2[previous*256 + i], max, max_length);

 nbparts[previous + (position)*256 +
     level*256*max_length] = out;
 return out;
}
```

This property means that it is possible to order the set of passwords whose Markov strength is less than a value and generate the nth password of this set. Algorithmic details are described in [5]. In order to use this technique for cracking passwords, we must:

- select the passwords to crack;
- select the maximum supposed length ($maxlen$) of the generated passwords;
- evaluate the cracking speed of all his processors, and the allowed crack time;
- multiply this cracking speed by the maximum crack time, and then divide by the number of different salts in order to calculate the number of passwords that could be tested in the selected amount of time:

  $N = crackspeed * cracktime/salts;$
- find the largest max value so that $nbp_{maxlen}('', 0, 0, max) < N$;
- generate all the candidate passwords and test them!

4.3 Applications

First of all, we have shown that passwords generated this way match the actual real world passwords (see Sect. 5.3). Moreover, Markov password generation is useful in several password cracking scenarios.

- Distributed password cracking: when distributing the password cracking work between nodes, it is necessary to divide the set of passwords in several subsets. It is of outstanding importance that the size of these subsets is known beforehand, to prevent one of the nodes from tackling a password set that is too large for its processing capabilities. In a perfect scenario, each nodes would be assigned with password indexes to delimit their subset, ie. node 1 would crack password 0 to password 99, node 2 from password 100 to password 199, etc. This is easy to achieve with Markov password generation as the nth password could be generated without the need to generate the $n - 1$ previous passwords.
- Rainbow tables: rainbow tables are a type of time-memory tradeoff that is very effective when applied to password cracking. It works by storing "chains" of pre-computed hashe, where the nth hash is $h_n = H(R_n(h_{n-1}))$. Here, H is the hashing function, and R_n is the "reduction function" that will transform an arbitrary value into a suitable password for step n. The "classic" reduction function is the base conversion function C, where $R_n(x) = C(x + n)$, and the destination base is the target character set. For example, the reduction function for alphabetic tables would convert numbers to base 26. It is possible to design an alternate function : if $G(x)$ generates the xth password from a Markov password set whose size is N, a suitable function could be $R_n(x) = G((x + n)\ mod\ N)$, where $a\ mod\ b$ is the remainder of the Euclidian division of a by b.
- Security consulting: being able to assign a strength (the Markov strength) to a password is the first step to magnificent Excel graphics and statistics that would look great in an audit report.

In the case of rainbow tables, Naranayan and Shmatikov [5] give a preliminary result: against 142 real users passwords, their attack, using Markov filtering and some regular expressions filters, recovers 96 passwords, against 39 passwords for RainbowCrack. Without much detail, it is hard to figure how good is this solution.

Using the RainbowCalc tool [4], we evaluated the effectiveness of the two methods. Let us suppose that RainbowCrack was very well optimized (12M MD4 hashes per second). The following results could be estimated:

Reduction function	Classic alphanum	Markov S = 350	Markov S = 300
Chain length	1E6	1E6	1E6
Chain count	4E8	1E7	8E6
Tables	10	10	7
Max length	8	14	14
Generation time (week)	551	20	11
Recovery rate	72.47%	86.57%	61.47%

The recovery rate has been estimated using results from Sect. 5.3. The classic reduction function should find 82.2% of passwords of size less than 9. However, only 88% of the real world passwords have such a short size, according to our sample.

5 Experimental results

In this section, we worked with a real world hashed password file of about 7,700 passwords. The passwords have been selected by French speaking users. They came in LM (LanMan, the original Microsoft password storage format) and NT hash flavour. Nearly 97% of the passwords have been cracked with the LM hashes using RainbowCrack (rainbow table cracking is described in [4]). Cracking effectiveness has been calculated based on the NT hash, which is stronger. Figure 1 shows the size distribution of the known passwords in this set. While the password set is large, it has been collected from a single source. The results might be dependent of "cultural" parameters specific of this source, and might not properly model the global problem.

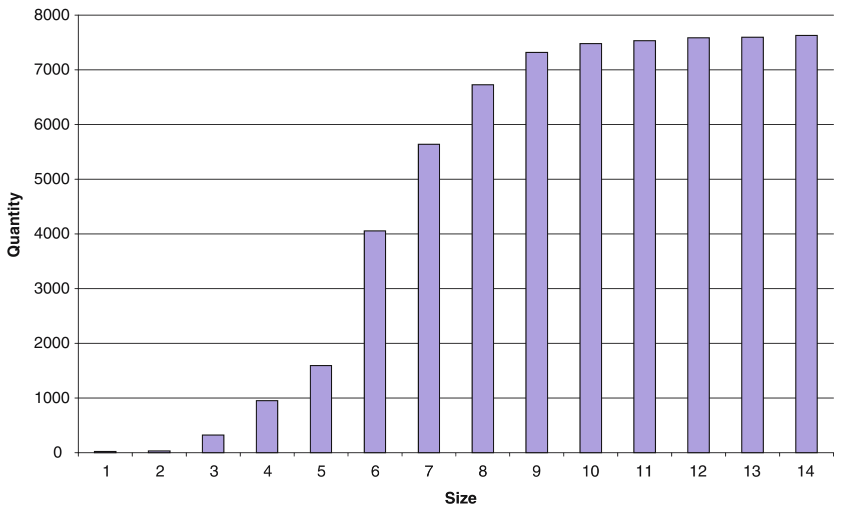


Fig. 1 Password size distribution

5.1 Tested implementations

5.1.1 *John the Ripper*

We used John the Ripper (JtR [6]) with unofficial contributions compiled in [7]. The contributions increase the cracking speed for NT hashes, making JtR the fastest general purpose MD4 cracker. We used it in several modes:

- "Standard mode": JtR tries passwords based on the users names, then uses a dictionary[3] finally, it runs its "incremental mode";
- "Incremental mode": JtR tries passwords based on a statistical method. An important limitation of this mode is that the generated candidate passwords length cannot exceed eight characters. By looking at Fig. 1 it is clear that 12% of the passwords are larger than eight characters, and are thus not crackable by the incremental mode. We used the standard "charset file" shipped with JtR, and a custom file generated for French passwords.
- "Stdin mode": JtR reads the candidate passwords from its standard input.

5.1.2 *Playstation 3*

We wrote a PS3 implementation of the NT hash function that could achieve a total speed of 180M pwd/s. It should be noted that this speed could only be achieved by brute forcing the first two characters. The other characters of the candidate passwords could be selected in an arbitrary way.

The implementation generates the candidates passwords on the PPC core, and sends them to the SPUs. The SPU brute-forces the first two characters of the candidate passwords. It should be noticed that a huge speedup could be gained by using the techniques presented in Sect. 2.1.2 and thus by improving the code. We believe that a theoretical speed of 280M MD4 pwd/s is achievable on the CELL processor.

[3] Transformation rules have been considered as well but they will be not described in this document.

5.2 Password generation process

5.2.1 *"Stupid" brute force*

This process works by trying incrementally all passwords: a, b, c, d, …, z, aa, ab, …. It is the easiest candidate passwords selection process, but obviously not the smartest. For every known (cracked by RainbowCrack) password in the test passwords set, the number of passwords that should be tested before it is recovered, has been computed. The PS3 figure of 180M pwd/s was considered in order to evaluate the time needed to find this password.

5.2.2 *JtR incremental mode*

The incremental mode of JtR has been tested for both the standard and custom "charset files".

5.2.3 *Markov*

We wrote a Markov password generator and benchmarked it by redirecting its output in JtR. The Markov strength of every known passwords has then been computed. Then for every values of the "max" parameter, the number of passwords cracked has been evaluated. The result is displayed in Fig. 2.

There is a significant gap at strength 350 for the standard Markov strength. It is caused by "company password", e.g. a password that is believed to be strong and that is used by many workers in a company. It might be also the default passwords that all accounts share when they are created. The

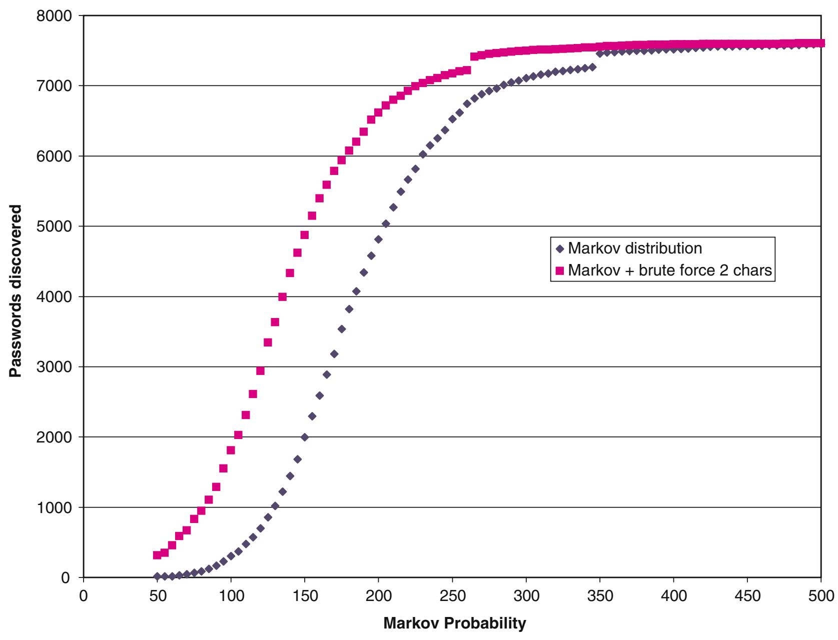


Fig. 2 Markov strength password distribution

cracking time needed for every target Markov strength has been evaluated using the previous benchmark.

5.2.4 Markov with first two characters brute-forced

This process works by generating passwords using the Markov password generator, and appending two character at their beginning. These characters have been tested in an exhaustive way (with a charset of 97 different characters, 97×97 passwords have been evaluated). The Markov strength of all known passwords without their first two characters has been evaluated.[4] This value has been used in order to evaluate the cracking time and the password recovery rate for every target Markov strength with respect to this mode. The PS3 figure of 180M pwd/s was used for these calculations.

5.3 Results

This section compares the cracking speed of JtR on a Xeon 2.4 GHz, our Markov password generator and an hypothetical brute forcer. The cracking speeds of a Markov password generator that brute-forces the first two characters and of a brute forcer both on a PS3 have been evaluated. Results are summarized in Fig. 3. It is not surprising that brute-force cracking should not be the preferred cracking method. Even with a vastly faster implementation (180M pwd/s vs. 6.4M pwd/s) it does not compete with JtR.

However, it is surprising to notice that the Markov password generator, while being far slower than JtR (1.3M pwd/s vs. 6.4M pwd/s) is actually performing better. After 22 h of cracking time, JtR recovered 6,362 passwords while our Markov generator recovered 6,745. This performance advantage is strengthened by the fact that it is easy to use distributed computing compared to the incremental mode of JtR. However, while JtR could be running forever and eventually find all passwords,[5] the Markov tools run for a predefined time. Subsequent runs would have to rehash previously tested passwords.

The different implementations that could be used for distributed password cracking have been compared. Brute-force attack has been ruled out because it was far from being as effective as the Markov password generation. JtR incremental mode is notoriously hard to distribute. It remains to consider the various Markov implementations, as shown in Fig. 4. It shows the password cracking rate in function of the cumulated cracking time in hours (on an AMD 3500+ if not specified, or a CELL processor for the PS3 variant). This demonstrates that someone (e.g. a virus writer who managed to steal encrypted passwords files by means of a malware) could break 98% of this company passwords in a week (168 h) by using 37 Keuros worth of PS3 hardware.

6 Conclusion

There are several classes of attacks against passwords. The most effective ones, used by malicious persons, are the

[4] We used the same statistical values than those previously exposed. For a better evaluation, statistical values should be calculated with a dictionnary where the first two characters of every word are truncated.

[5] …whose length is less than 9 and with ascii characters!

Fig. 3 Comparison of several implementations

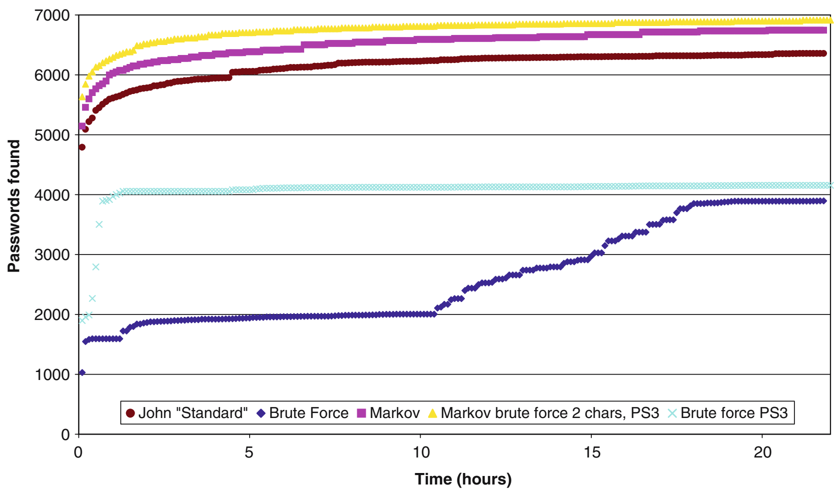
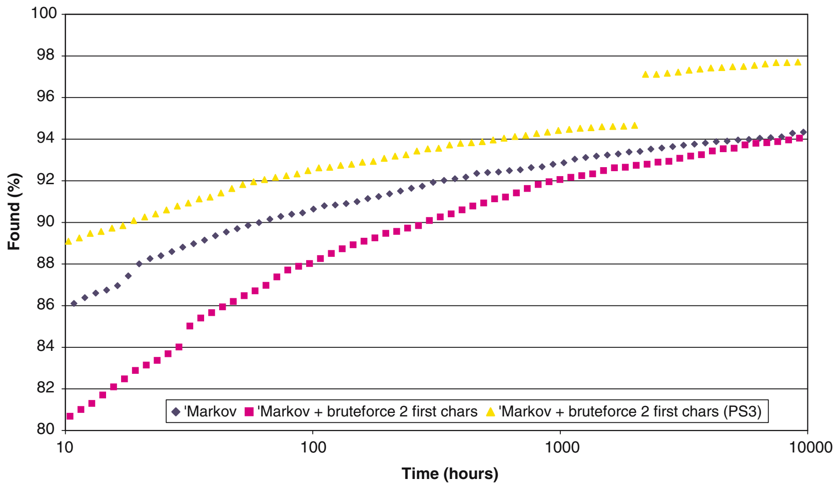


Fig. 4 Comparison of distributed implementations



simplest: phishing, keylogging, educated guesses, etc. However, where the malicious person or the penetration tester only needs a single password, the security consultant craves for the whole list of plaintext passwords when performing an audit.

In this paper, several enhancements that could be included in password cracking tools are described. These enhancements could dramatically speed up the cracking process. However, we believe that future research should be undertaken in order to evaluate more precisely the effectiveness of the Markov based tools.

Markov filter based techniques could be deployed in an effective way by malicious code (virus, worms, etc.) and create a large distributed password cracking network. Centralized coordination is not mandatory, but the malicious payloads should be able to publish their results, should a password be cracked. Botnets are however a better solution for this application. "Strong" passwords protecting network ressources, such as WPA or IPSEC pre-shared keys, could be tempting enought for a botnet master. A typical WPA-PSK cracker should reach 100 keys/s on typical hardware. An obfuscated version (such as defined in [8]) could covertly (using techniques such as described in [9]) crack 20 keys/s. A botnet of 20,000 computers could test 400,000 pwd/s, cracking any password whose Markov strength is less than 247 (80% of passwords in our sample) in 24 h.

The Markov strength concept could be included in operating systems in order to implement a password checking policy that would be far more effective than those currently in force (typically, rules like "use at least eight characters, with at least a character in each of the categories: alphabetic, numeric, symbol" are not effective). It also provides a convenient metric for the security professional.

References

1. Project RainbowCrack http://www.antsight.com/zsl/rainbowcrack/
2. The OpenCiphers Project http://openciphers.sourceforge.net/oc/
3. Copacobana, a Codebreaker for DES and other Ciphers http://www.copacobana.org/
4. Oechslin, P.: Making a Faster Crytanalytical Time-Memory Trade-Off (Advances in Cryptology - CRYPTO 2003, In: 23rd Annual

International Cryptology Conference, Santa Barbara, California, USA, August 17–21, 2003, Proceedings. Lecture Notes in Computer Science 2729 Springer 2003, ISBN 3-540-40674-3)
5. Narayanan, A., Shmatikov, V.: Fast Dictionary Attacks on Passwords Using Time-Space Tradeoff The University of Texas at Austin http://www.cs.utexas.edu/~shmat/shmat_ccs05pwd.pdf
6. John the Ripper password cracker, http://www.openwall.com/john/
7. John the Ripper unofficial contributions, http://www.banquise.net/misc/patch-john.html, http://btb.banquise.net/bin/myjohn.tgz
8. Beaucamps, P., Filiol, E.: On the possibility of practically obfuscating programs towards a unified perspective of code protection. J. Comput. Virol. Vol. 3, Number 1/April, 2007
9. Tsafrir, D., Etsion, Y., Feitelson, D.G.: Secretly monopolizing the CPU Without Superuser Privileges, http://www.cs.huji.ac.il/~dants/papers/Cheat07Security.pdf