An Architecture for Distributed Dictionary Attacks to OpenPGP Secret Keyrings

Massimo Bernaschi
IAC-CNR
Viale del Policlinico, 137, Rome, Italy
massimo@iac.rm.cnr.it

Mauro Bisson
CASPUR
Via dei Tizii, 6, Rome, Italy
m.bisson@caspur.it

Emanuele Gabrielli
Dip. di Informatica e Scienze dell'Informazione
Università di Genova, Italy
gabrielli@disi.unige.it

Simone Tacconi
Ministero dell'Interno
Servizio Polizia Postale e delle Comunicazioni
simone.tacconi@interno.it

Abstract

We describe a distributed computing platform to lead large scale dictionary attacks against cryptosystems compliant to OpenPGP standard. Moreover, we describe a simplified mechanism to quickly test passphrases that might protect a specified private key. *Only passphrases that pass this test complete the full (much more time consuming) validation procedure. This approach greatly reduces the time required to test a set of possible passphrases.*

1 Introduction

A dictionary attack is a technique for defeating a cryptographic system by searching its decryption key or password/passphrase in a list of words or combinations of these words. Although it is widely accepted that the main factor for the success of a dictionary attack is the choice of a suitable list of possible words, the efficiency and reliability of the platform used for the attack may become critical factors as well. Hereafter, we present a distributed architecture for performing dictionary attacks that can exploit resources available in local/wide area networks (in P2P style) by hiding all details of communication among participating nodes. As an example of possible cryptographic challenge for which the platform can be used, we selected the decryption of the private *keyring* of the GnuPG software package. From this viewpoint, the present work can be considered a replacement and an extension of *pgpcrack* (that is no longer available), an utility used for cracking PGP encrypted files. Note that the structure of the OpenPGP *secring* is much more complex with respect to the original PGP. To the best of our knowledge, no equivalent *fast* cracking system exists for OpenPGP. Other scalable distributed cracking systems were proposed in [3] and [9]. Due to the lack of space we can not present a detailed comparison, but we just mention that our system pays much more attention to reliability and portability issues than the cited systems. The paper is organized as follows: Section 2 describes the features of OpenPGP, the standard to which GnuPG makes reference; Section 3 describes our approach to the attack of the GnuPG *keyring*; Section 4 introduces the architecture we propose for the distributed attack; Section 5 gives some information about the current implementation; in Section 6 we present some preliminary results and, finally, Section 7 concludes with future perspectives of this activity.

2 OpenPGP Standard

OpenPGP is a widely used standard for encryption and authentication of email messages. It is defined by the OpenPGP Working Group in the Internet Engineering Task Force (IETF) Proposed Standard RFC 2440 [5]. OpenPGP derives from PGP (Pretty Good Privacy), a software package created by Phil Zimmermann in the beginning of nineties. GnuPG [2] is a well-known example of software package compliant to OpenPGP standard available in the public domain. New commercial versions of PGP are also compliant to OpenPGP standard.

The OpenPGP standard adopts a hybrid cryptographic scheme. For instance, message encryption uses both symmetric and asymmetric key encryption algorithms. The sender uses the recipient's public key to encrypt a shared key (i.e. a secret key) for a symmetric algorithm. That key is used to encrypt the plaintext of the message. The recipient of a PGP encrypted message decrypts it using the session key for a symmetric algorithm. The session key is included in the message in encrypted form and it is decrypted in turn by using the recipient's private key. These keys are stored in

Proceedings 21st European Conference on Modelling and Simulation
Ivan Zelinka, Zuzana Oplatková, Alessandra Orsoni ©ECMS 2007
ISBN 978-0-9553018-2-7 / ISBN 978-0-9553018-3-4 (CD)

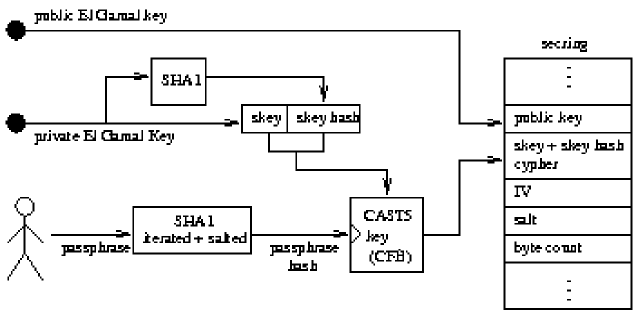

Figure 1. OpenPGP *keyring*

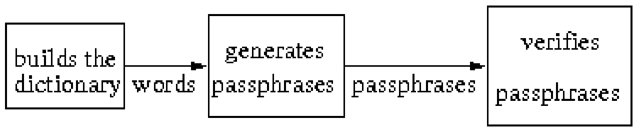

Figure 2. The three phases of the attack

two separate data structures, called "keyrings": private keys in the private keyring, public keys in the public keyring. Every keyring is a list of records, each of which associated to a different key. In order to prevent disclosures, private keys are encrypted with a symmetric algorithm, by using a hash of a user-specified passphrase as secret key. For what concerns GnuPG, as shown in Figure 1, the asymmetric encryption algorithm is El Gamal [7], the hash algorithm is SHA1 [6] and the symmetric encryption is CAST5 [4], used in CFB mode [5].

3 Attack Strategy

One of the most critical issues regarding OpenPGP security is the secrecy of passphrases protecting private keys. The knowledge (by any means achieved) of the passphrase allows a malicious user to perform critical operations as signature and decryption of messages belonging to the legitimate owner of the passphrase. For this reason, the goal of the attack is to find the passphrase, starting from a private keyring in OpenPGP format. The attack is divided in three phases, each of which receives as input the output of the preceding step, as shown in Figure 2.

The first phase is devoted to build the dictionary used for passphrases generation that represents the second phase. The third phase consists of the test of every generated passphrase.

3.1 Dictionary Compilation Phase

In this phase, the basic dictionary is created starting from a set of text files. This a pretty simple procedure: each different word is placed in the list that constitutes the dictionary. In order to increase chances of success, the content of these text files should contain information somehow related to the legitimate owner of the passphrase under attack. This process is depicted in Figure 3.

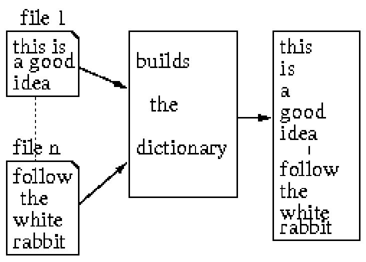

Figure 3. The first phase: the build of the dictionary

3.2 Passphrase Generation Phase

This second phase produces a list of passphrases by applying a set of generation rules to all words found in the dictionary. Every rule involves the current word and a chosen number of subsequent words and allows for the generation of passphrases, by performing permutations of the order of words and/or substitutions of single characters. In this way, the obtained passphrases are reasonably compliant with rules of natural language.

For instance, if we apply rules that involve a word and four subsequent words to generate passphrases with a length ranging from one to five words, for each word in the dictionary we obtain 39 possible passphrases:

- the current word as in the dictionary, then the same word with all lower case letters and all upper case letters (3 passphrases).
- the current word and the following one, taken in the original order and in the reverse order, with all lower case letters, all upper case letters and the unmodified case (6 passphrases).
- all possible permutations of the current word and the two subsequent words, with all lower case letters and all upper case letters (18 passphrases).
- the current word and the three subsequent words, taken with the order and the case in the dictionary and in reverse order, with all lower case letters and all upper case letters (6 passphrases).
- the current word and the four subsequent, taken with the order and the case in the dictionary and in reverse order, with all lower case letters and all upper case letters (6 passphrases).

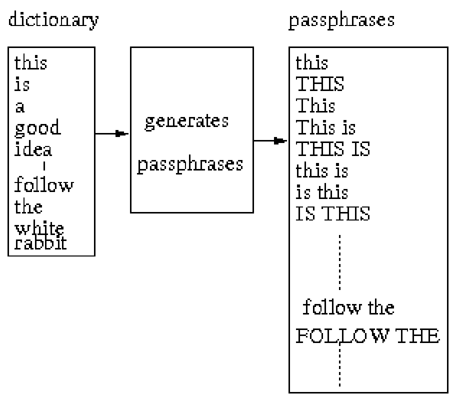

Figure 4. The second phase: generation of passphrases.

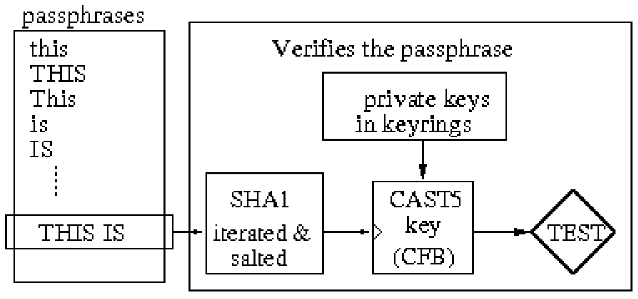

Figure 5. The third phase: verification of the passphrase.

Note that in the generation of passphrases with four and five words, some permutations are not considered, since they yield sequences unlikely for human memorization. The generation phase is depicted in Figure 4.

3.3 Passphrase Verification Phase

This phase is the *core* of the attack and the most expensive from the computational point of view. Each passphrase generated in the previous phase is checked by following an *incremental* approach aimed at minimizing the cost of the controls required by the OpenPGP standard. For this reason, a symmetric key for CAST5 algorithm is derived from every passphrase, by applying the SHA1 algorithm in iterated and salted mode. Such a key is used to try a decryption of encrypted components relating to private key. This process is represented in Figure 5.

In order to check whether the passphrase under test is the right one, it is necessary to verify the plaintext obtained from the decryption procedure. This operation is performed taking into account how the OpenPGP standard represents components relating to private keys in keyrings.

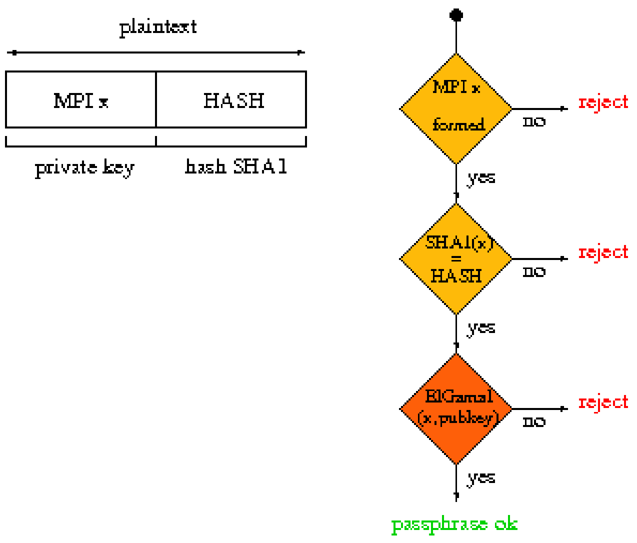

Figure 6. Validation test

As shown in Figure 6, a private key is represented as a Multi Precision Integer (MPI), followed by its hash, computed with SHA1. For this reason, in the first step we verify if the left part of plaintext is a well-formed MPI. In case of success, we double check whether the result of SHA1 applied to the MPI matches with the hash found in the plaintext. Only for those passphrases that pass this second test, we control the fulfillment of the algebraic relationship that should be between the MPI and the corresponding public key. If this final check is successful, then the passphrase under test is the correct one. Note that the first two controls have a low computational cost but they may produce *false positives*. The last control is *exact* but it is very expensive from the computational viewpoint. GnuPG does not carry out the first control that is already very selective. By following this multi-step procedure our validation test is much more effective.

4 Distributed Architecture

The attack described in the previous section has been deployed over a loosely coupled distributed architecture. The three phases of the attack are scattered over the nodes of this network. There is a main node and two different groups of peers that share their computational resources.

4.1 General Requirements

Since this network has been conceived to work with heterogeneous systems in a geographic context, the proposed architecture guarantees the following requirements:

scalability: the number of network nodes can be increased,

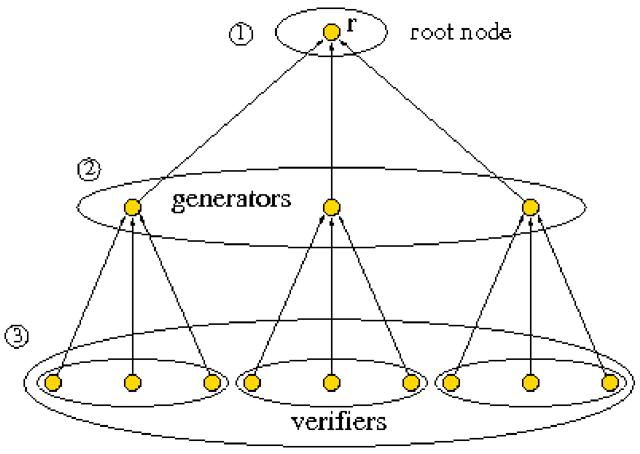

Figure 7. Nodes organization.

augmenting the overall computational power.

load balancing: the computational load must be distributed among nodes in proportion to their capabilities, so that to avoid local starvation.

flexibility: since the availability of each node in the network is aleatory, the architecture must be able to adapt itself to variations of available resources by changing the distribution of charge.

fault tolerance: possible failures of a node must not subvert the overall computation, thus the system must able to re-assign the workload and to resume local computation.

4.2 Overall Organization

The proposed architecture consists of three levels, each of which implements a specific phase of attack, as represented in Figure 7. Each level receives information from the upper level, elaborates them and then supplies the lower level.

The first level is constituted by a single "root" node, denoted as r, that is responsible for the compilation of the dictionary. The second level consists of a variable number of nodes, named "generators" and denoted as g, that form the "generation network", indicated as G. Such a network is devoted to generation of passphrases starting from the dictionary compiled in the first phase by the "root" node. The third level consists of a variable number of nodes, named "verifiers" and denoted as v, that form the "verification network". Such a network is in charge of verifying whether any of the generated passphrases decrypt the private key given in input. Node r and the sets of nodes G and V form the network system $\sum = < r, G, V >$.

System $\sum$ has a tree-like topology where generator nodes play the role of children of root node r. Verifier nodes v are divided in groups, each of which is assigned to a generator node g, as depicted in Figure 7. Each node acts as client with respect to the father node and as server with respect to the child node.

Every node performs a specific task:

- root node r compiles the dictionary D, divides it in partitions $P_i(D)$ and assigns the i^{th} partition to the generator node g_i;
- each generator node g_i extracts from $P_i(D)$ a list of passphrases L and divides it in partitions $P_j(L)$. Every partition $P_j(L)$ is assigned to a verifier node v_j^i (where the superscript i indicates that v_j is a child of g_i);
- each verifier node v_j^i checks all passphrases in the assigned partition $P_j(L)$ with respect to the private key provided in input.

This model of interaction, represented in Figure 8, makes easier to achieve a reasonable load-balancing by assigning more work to groups with more verifier nodes. Every node of the network needs to know only the identifier of its father node, of the "root" node and of all its child nodes (if any), in order to communicate with them. Moreover, for each of its child nodes, a father node checks the status of available resources and stores the last messages sent to it. Information stored in a node are maintained until child nodes do not confirm the completion of operations assigned to them. Note that child nodes never communicate each other.

Communication occurs by means of message exchanges that require receiver's confirmation. A node only accepts messages coming from the father, its children and, possibly, the root. Messages can be grouped as follows:

task messages: used to exchange information about the attack.

maintenance messages: used for handling asynchronous events of the network.

heart-beating messages: aimed at detecting failures and sending information about available resources.

4.3 System Life-cycle

An instance of the system begins with just the root node. As new nodes join the network to participate in the attack, (this is done by sending a message to the root node), generation and verification networks are populated. A new node is assigned to the generation network if there are no generator nodes (this is the typical situation in the beginning), or if all existing generator nodes serve the maximum number

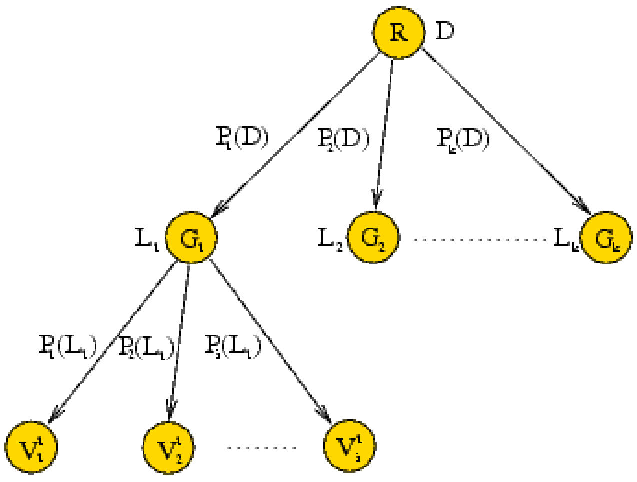

Figure 8. Interaction among nodes

of verifier nodes (this maximum number can be tuned at run time). Otherwise, the new peer becomes a verifier node and it is assigned as child to the generator node having the lowest number of children. The expansion model of the system is shown in Figure 9.

An instance of the system ends when the correct passphrase for the given private key is found. The verifier node on which a candidate passphrase passes successfully the first two controls described in section 3.3 sends it to its father generator node. This node performs further controls (the final test described in section 3.3) and, on success, then forwards the passphrase to the root node that, as a consequence, stops the system. This process is depicted in Figure 10. For what concerns the single nodes, every peer can be in one of the following states:

running: the node is performing its own task;

serving: the node is executing the assigned task and performing a maintenance operation that involves one or more child nodes;

stopped: the node is not executing a task because it is involved in a maintenance operation launched by its father node or by itself;

The root node can be in either running or serving state, a generator node can be in running, serving or stopped state, a verifier node can be in either running or stopping state. State transitions occur when a message is received, or as a consequence of a local event.

Each node is able to produce local events that are handled by executing maintenance operations. Nodes generate events that are compatible with their current state. An event triggers a transition in a state where the corresponding maintenance operation must be executed. Three kinds of events are possible:

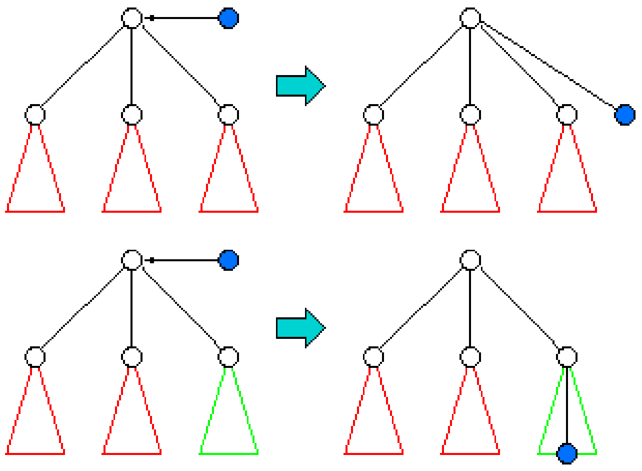

Figure 9. Propagation scheme

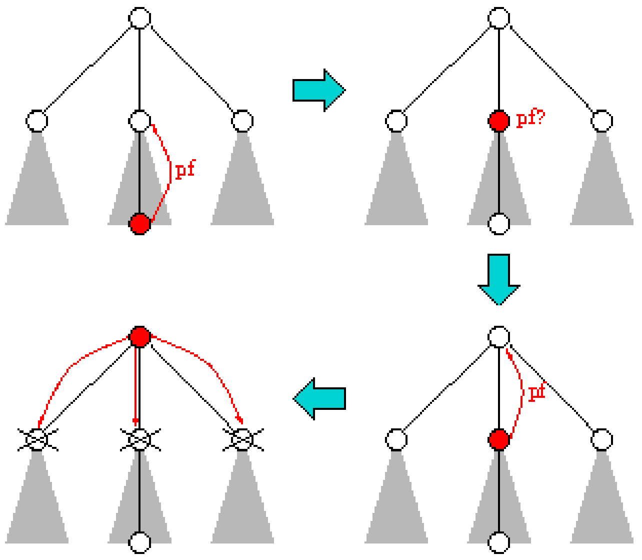

Figure 10. Shutdown scheme

soft-quitting (SQ): produced when a node explicitly leaves out the system;

hard-quitting (HQ): generated when a node detects an unexpected quitting of a child, for example due to a child failure.

swapping (SW): event that occurs when a node exchanges its role with a child.

Each node manages its soft-quitting related operations and hard-quitting related operations of its children. Verifier nodes, since do not have children, do not need to manage hard-quitting and swapping events. Finally, the root node can not swap its role with a child.

If the root node quits the network, the entire instance of the system halts. When a failure (HQ operation) occurs in a generator node, the root appoints one of the orphan verifier nodes as new generator node for the remaining orphans nodes and assigns to it left pending partitions by the failed generator node. When a generator node wants to quit the system (SQ operation), it elects a substitute, choosing it among its child (verifier) nodes, and supplies to it all the information required to complete the task. Finally, the outgoing node informs the root node and quits.

When a failure occurs in a verifier node (HQ operation), the father generator node forwards to other child nodes the pending list of passphrases previously assigned to the broken node and informs the root. When a verifier node wants to quit the system (SQ operation), it informs its father generator node about the number of checked passphrases in the pending list. The generator node then supplies residual passphrases to its other child verifier nodes and informs the root note. Finally, generator nodes, in case of variation of their own resources with respect to those available to child verifier nodes, may swap their role with one of the child verifier nodes (SW operation), in order to assign to the verification network the most performing nodes.

5 Implementation

The system has been implemented in a single application, named *dcrack*, that is able to perform all the three phases of the attack, *i.e.*, dictionary compilation, passphrase generation and passphrase verification. In such a way, the same application runs on every node of system. The code has been implemented in ANSI C taking into account the requirement of being usable in a multi-platform environment. To this purpose, the application relies only on portable components as shown in Figure 11. In particular, the Apache Portable Runtime (APR) [1] is a set of APIs that guarantees software portability across heterogeneous platforms, through a replacement of functions that are not supported in the underlying operating system. For instance, the use of the APR environment allows to exploit synchronization mechanisms like the "condition-variables", unavailable in the native Windows environment. As to the networking issues, we resorted to the MIDIC middleware, a software layer that provides advanced communication services. MIDIC, in turn, relies on the JXTA technology [8] in order to enable communication between P2P applications, independently of network complexity. For example, if a node accesses to the network through a firewall that enables only HTTP traffic, the middleware automatically establishes a HTTP tunnel in order to guarantee reachability. Both MIDIC and JXTA exploits the APR environment.

The application is subdivided in components, each of which implements a specific function in the node where it runs. The subdivision is made on the basis of a logical classification of activities common to all nodes:

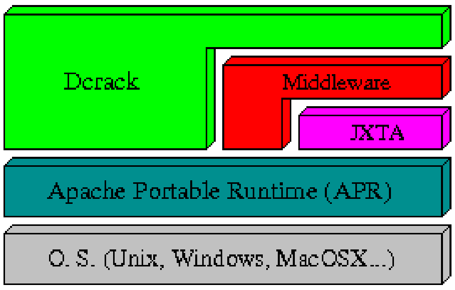
Dcrack
Middleware
JXTA
Apache Portable Runtime (APR)
O. S. (Unix, Windows, MacOSX...)

Figure 11. Software structure

execution of task: each task is made of components, the *worker* that acquires and processes information about the attack and *server* that makes available results of required computations;

maintenance operations: such operations are managed by a *controller* component, for what concerns quitting the system and failures, and by a *recruiter* component for the entry of new nodes in the system.

heart-beating activity: this activity is carried out by a *beater* component for sending heart-beating messages to child nodes and by a *Heart* component for receiving such messages.

Active components of the application for the three classes of nodes and communication flows between them are shown in Figure 12. Task messages are sent from the

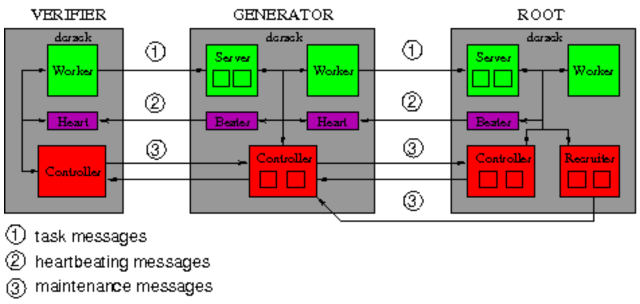

Figure 12. Communication scheme

Work component to the Server component of the application running on the father node. Heart-beating messages are sent from the Beater component of the application running on the father node to the Heart component running on the child nodes. Maintenance messages are exchanged between the controller component of the application running on a child node and the corresponding component in the father node.

Each component runs in a thread whose implementation depends on the platform (but this is transparent to the application since it relies on the APR environment). Cooperation between threads follows the *work-crew* model. Moreover, threads in charge of components that may require simultaneous communication (*i.e.,* the server, controller and recruiter components) generate a *service* thread to which the communication is demanded. Cooperation between component threads and service threads follows the *boss-worker* model.

6 Experimental Results

We measured the performances of the proposed architecture in a test-bed constituted by a 100baseT Ethernet LAN with 20 personal computers, equipped with a 2.8 Ghz Intel Pentium IV processor and 512Mb of RAM running the Linux operating system. As sample target of the attack, we selected the GnuPG cryptographic software with an ElGamal key having a length of 768 bits.

To generate the dictionary we started from the text of "Divina Commedia" (a famous epic poem of Italian literature) and, as a consequence, generated passphrases are in Italian. In order to evaluate the throughput of the system we chose a passphrase that could not be found with this dictionary, forcing the system to generate and test all passphrases that could be derived from the input text and the defined passphrase generation rules.

Before starting the full experiment, we carried out some preliminary tests, in order to find out how many verifier nodes could be fed by a single generator node. Therefore, the following parameters have been evaluated: k, the number of passphrases that can be checked by a verifier node in a second; t_g, the time required to a generator node to generate k passphrases; t_s, the time required to a generator node to compress and send k passphrases to a verifier node; Our tests showed that a verifier node is able to check about 1000 passphrases per second. A generator node requires 0.6 ms to generate 1000 passphrase and about 10ms to compress and send them. Thus, a generator node needs about 11ms to set up the workload that a verifier node carries out in one second. As a consequence, the adequate ratio R between the number of generator nodes and verifier nodes is given by:

$$R = 1/(t_g + t_s) = (1/0,011) \sim 90$$

In other words, with these settings, each generator node could feed up to 90 verifier nodes.

In the test environment, we used a variable number of nodes but, since the time required to generate, compress and send passphrases is about two orders of magnitude smaller than the time used for verification, in all tests, we used a single generator task that coexisted with the root task on a single node (same computer) of the network.

The results we obtained are very encouraging, since the throughput of the system (measured as the inverse of the time required to test all possible passphrases) increases in a linear way with respect to the number of verifier nodes. We compared these results with those produced by a commercial solution for Grid computing (AGA by Avanade) and found that the throughput of our solution is (about) 20% higher (obviously with the same number of nodes). Finally, no appreciable difference has been found with respect to the operating system of the nodes (PCs we used were dual-boot, so we could test Linux and Windows on the same hardware). As to the reliability of the platform, we checked it in a separate set of tests in which we used up to three generator nodes. Failures of both verifier and generator nodes were successfully managed by the infrastructure by following the procedures described in section 4.3.

7 Conclusions

We presented an architecture to perform distributed dictionary attacks. The system has been tested on a *private keyring* of the GnuPG cryptosystem after a careful study of the features of the encryption system. In particular we devised a technique to quickly check candidate passphrases by limiting the execution of the most expensive control to a subset of the passphrases selected according to much less expensive controls. There are a number of possible directions for future activities. For instance taking into account the results reported in section 6, it is possible to introduce new generation rules and increase their complexity. Besides that, since there are many applications with features similar to cryptographic challenges, another interesting possibility could be to employ the distributed architecture for other computational tasks.

References

[1] Apache portable runtime (apr). http://apr.apache.org/.

[2] The gnu privacy guard (gnupg). http://www.gnupg.org/.

[3] Medussa. http://www.bastard.net/∼kos/medussa/medussa.html.

[4] Rfc2144: The cast-128 encryption algorithm. http://www.faqs.org/rfcs/rfc2144.html.

[5] Rfc2440: Openpgp message formats. http://www.faqs.org/rfcs/rfc2440.html.

[6] Rfc3174: Us secure hash algorithm 1 (sha1). http://www.faqs.org/rfcs/rfc3174.html.

[7] T. E. Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, IT-31(4):469–472, 1985.

[8] L. Gong. Jxta: a network programming environment. *Internet Computing, IEEE*, 5(3):88–95, May/June 2001.

[9] T. Perrine and D. Kowatch. Teracrack: Password cracking using teraflop and petabyte resources. *San Diego Supercomputer Center Security Group Technical Report*, 2003.