An Architecture for Distributed Dictionary Attacks to Cryptosystems

Massimo Bernaschi
IAC-CNR
Viale del Policlinico, 137, Rome, Italy
Email: m.bernaschi@iac.cnr.it
Mauro Bisson, Emanuele Gabrielli
Dipartimento di Informatica
Università "La Sapienza", Rome, Italy
Email: {bisson, gabrielli}@di.uniroma1.it
Simone Tacconi
Ministero dell'Interno
Servizio Polizia Postale e delle Comunicazioni
Email: simone.tacconi@interno.it

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

Index Terms— **OpenPGP, Volunteer computing, Middleware.**

I. 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 a private *keyring* generated by 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. Note that the structure of the GnuPG *secring* is much more complex with respect to the original PGP. To the best of our knowledge, no equivalent *fast* cracking system exists for GnuPG. Other scalable distributed cracking systems were proposed in [1], [2] and [3]. We briefly describe some of their features in section VI.

The rest of the paper is organized as follows: Section II describes the features of OpenPGP, the standard to which GnuPG makes reference; Section III describes our approach to the attack of the GnuPG *keyring*; Section IV introduces the architecture we propose for the distributed attack; Section V gives some information about the current implementation; in Section VI we present some preliminary results and, finally, Section VII concludes with future perspectives of this activity.

II. 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) Standard RFC 4880 [4]. OpenPGP derives from PGP (Pretty Good Privacy), a software package created by Phil Zimmermann in the beginning of nineties. GnuPG [5] is a well-known public domain software implementation of the OpenPGP standard. New commercial versions of PGP are also compliant to the 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 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

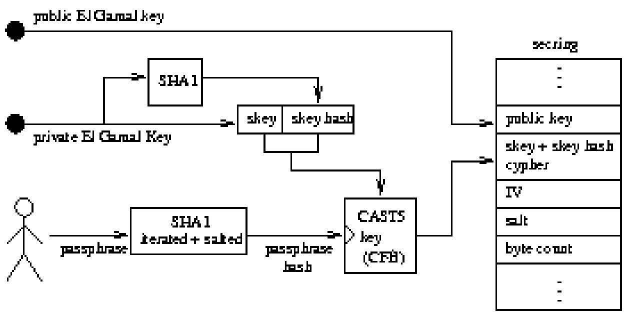

Figure 1. OpenPGP *keyring*

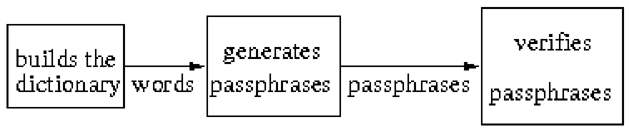

Figure 2. The three phases of the attack

GnuPG, as shown in Figure 1, the asymmetric encryption algorithm is El Gamal [6], the hash algorithm is SHA1 [7] and the symmetric encryption is CAST5 [8], used in CFB mode [4].

III. 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 gives the chance to a malicious user to execute critical operations as signature and decryption of messages belonging to the legitimate owner of the passphrase. For this reason, the attack to the OpenPGP system aims at finding the passphrase associated to a private keyring stored according to the 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 during the second phase in which there is the generation of the passphrases. The third phase consists of the test of every generated passphrase.

A. Dictionary Compilation Phase

In this phase, the basic dictionary is created starting from a set of text files. The procedure is quite simple: 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.

B. 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 the generation of passphrases, by performing permutations of

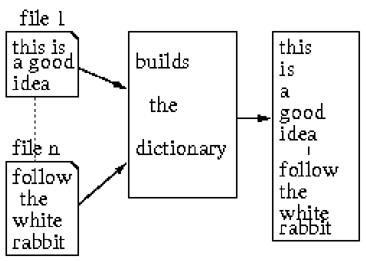

Figure 3. The first phase: the build of the dictionary

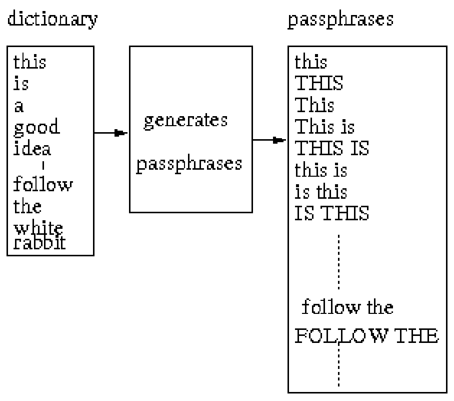

Figure 4. The second phase: generation of passphrases.

the order of words and/or substitutions of single characters. In this way, the obtained passphrases are reasonably compliant with the basic rules of a 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).

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.

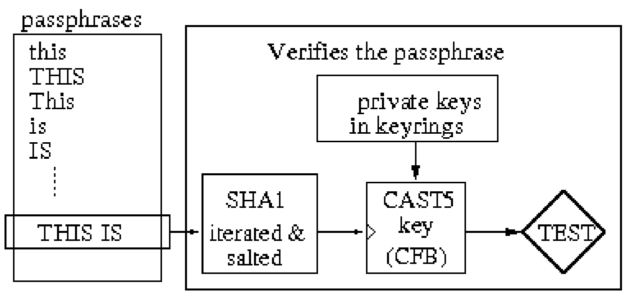

Figure 5. The third phase: verification of the passphrase.

C. 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.

As shown in Figure 6, a private key is represented by a Multi Precision Integer (MPI), followed by its SHA1 hash. For this reason, in the first step we verify if the left part of plaintext is a well-formed MPI. An MPI consists of two parts: a two-byte scalar that is the length of the MPI in bits (starting from the most significant non-zero bit) followed by a string of bytes that contain the actual integer. For instance, the string [00 09 01 FF] is a well-formed MPI with the value of 511 whereas the string [00 0A 01 FF] is not a valid MPI. If the plaintext is composed of l bytes ($l \geq 23$, since 20 bytes are required for the SHA1 hash and at least 3 bytes for the MPI) we have:

$$[b_l b_{l-1} | \, b_{l-2} ...] \, [b_{20} b_{19} ... b_1]$$

The first test consists in the verification that the two-byte scalar $b_l b_{l-1}$ represents the bit length of the string $b_{l-2}...b_{21}$. Since the size l of the plaintext is known (it equals the size of the ciphertext) this test only requires the verification of the relation:

$$b_l b_{l-1} = 8 - l_z(b_{l-2}) + (l - 23) * 8$$

where $l_z(b_{l-2})$ is the number of leading zeros in byte b_{l-2}. If the first test succeeds, we double check whether the result of SHA1 applied to the MPI $b_l b_{l-1}...b_{21}$ matches the hash found in the plaintext, $b_{20}b_{19}...b_1$. Only for those passphrases that pass this second test, we control the fulfillment of the correct algebraic relationship between the MPI and the corresponding public key. If this

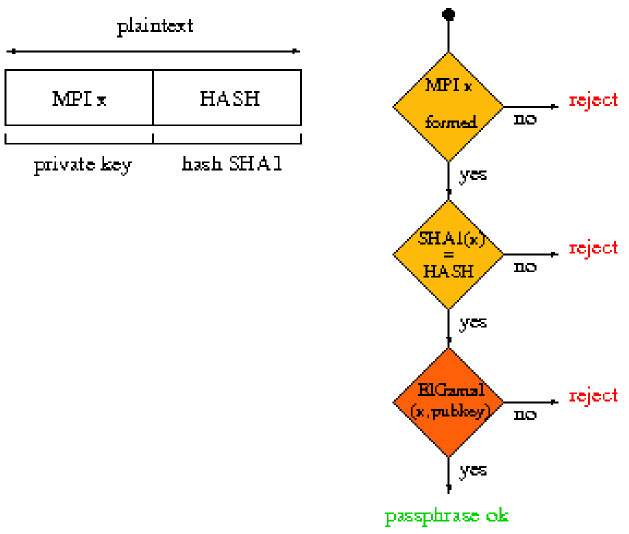

Figure 6. Validation test

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 cost/effective.

IV. 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.

A. 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 easily increased, augmenting the available computational power.

load balancing: the computational load must be distributed among nodes according to their capabilities, so that to prevent local starvation.

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

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

B. 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

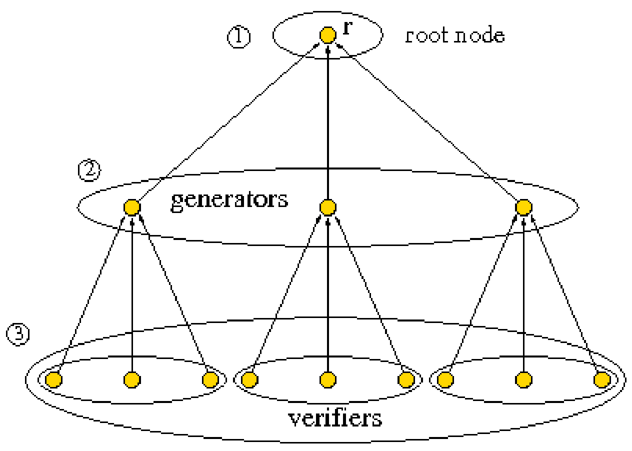

Figure 7. Nodes organization.

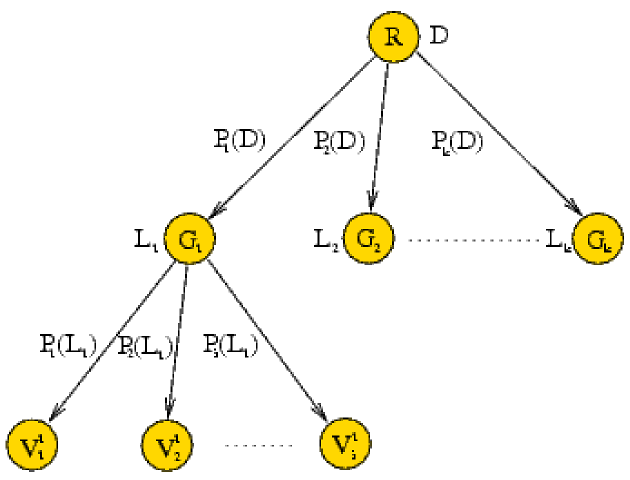

Figure 8. Interaction among nodes

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" G. Such a network is devoted to the 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 parent node and as server with respect to any 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^i_j (where the superscript i indicates that v_j is a child of g_i);
- each verifier node v^i_j 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 parent 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 parent 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 messages that require an explicit receiver's confirmation. A node only accepts messages coming from the parent, 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 related to the network.

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

C. 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 already the maximum number 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 III-C sends it to its parent generator node. This node performs further controls (the final test described in section III-C) and, on success, finally 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:

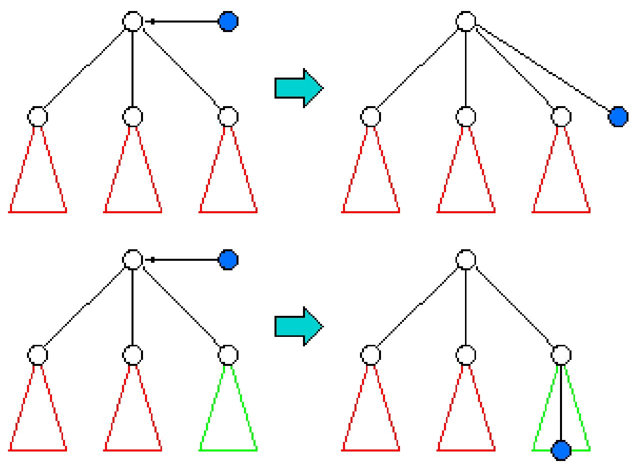

Figure 9. Propagation scheme

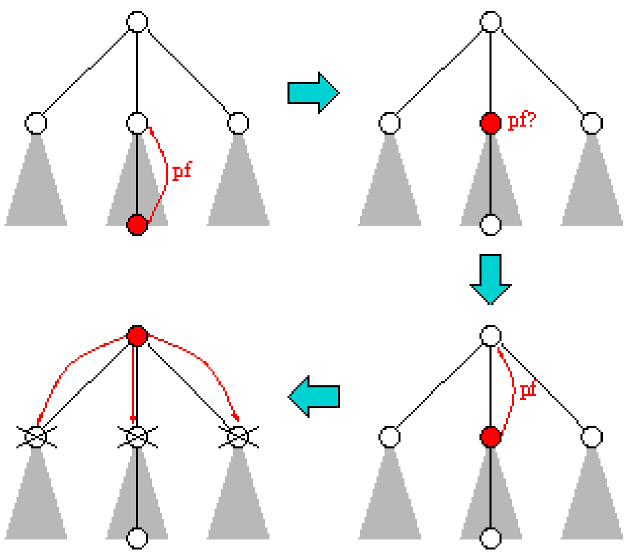

Figure 10. Shutdown scheme

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 parent 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.

Nodes execute maintenance operations when local events take place. The events must be compatible with the current state of the nodes. Usually, an event triggers a transition in a state where the corresponding maintenance operation is carried out. Three kinds of events are possible:

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. Moreover, the root node can not swap its role with a child.

If the root node suddenly quits the network, the entire instance of the system halts, unless the root node explicitly migrates all the information related to the instance to another node that becomes the new root. A possible future extension of the present architecture is to provide support for "multiple" roots that are automatically updated and may substitute, in a transparent way, the original root in case of failure.

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 pending partitions left by the failed generator node. When a generator node wants to quit the system (SQ operation), it elects a substitute, choosing it among its 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 parent 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 parent generator node about the number of checked passphrases in its 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.

V. 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 12. In particular, we use the Apache Portable Runtime (APR) [9], a set of Application Programming Interfaces (API) 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", available in the Windows environment only with the latest versions. To simplify communication operations and the management of the tasks life-cycle, we resorted to the MIDIC middleware that we briefly describe in the following section.

A. *MIDIC*

MIDIC is a middleware [10] that allows to set up a *computing infrastructure* composed by the aggregation of practically independent *institutions* (*e.g.* Research Laboratories, Universities). Each institution provides all required resources (*e.g.*, servers and network bandwidth) and services (*e.g.*, job scheduling and user authentication) to support computing activities of its users. Institutions store their public data in *Domain Name System* (DNS) tables, thus allowing other institutions to discover and interact with them. The infrastructure supports institutions interested in the execution of *computing applications* which require many CPU cycles and operate in contests where: *i)* safety and privacy are not binding factors; *ii)* the low cost of the solution is a factor of outstanding importance, and *iii)* high technical proficiency for systems management and applications development is not necessary. The MIDIC infrastructure supports applications that can be organized according to the *master-worker* model. Systems belonging to the infrastructure can act as *masters*, by publishing jobs and tasks, or as *workers* by running tasks. Tasks published by a master can be assigned to workers of external institutions.

The infrastructure supports both master and workers for issues related to: *i)* application cycle management; *ii)* fair distribution of the resources among the applications (to prevent some applications from monopolizing donated resources), and *iii)* exchange of data and messages.

MIDIC includes two components: a *server MiddlewareEndPoint* and a *user MiddlewareEndPoint*. The server MiddlewareEndPoint is the server side component which runs on institutions servers and provides user MiddlewareEndPoints and other server MiddlewareEndPoints with required institution services. The user MiddlewareEndPoint is the client side of the middleware. It runs on users machines (usually personal computers) and provides applications with API to access middleware services. To satisfy the main goal to support a broad range of applications, and since it runs on users machines, it has been implemented so that it: *i)* is portable on major Operating Systems (*e.g.* Windows, Linux and MacOS), *ii)* supports applications written in different programming languages (*e.g.,* C, C++, JAVA, Perl and Python), *iii)* allows applications to communicate across firewalls, and *iv)* reduces the impact of unpredictable user actions (*e.g.,* sudden reboot or power-off). The portability requirement is satisfied by resorting to the APR library. Moreover, the API is implemented as Remote Procedure Calls (RPC) by using a combination of HTTP and XML. In this way, the API supports any language that can handle TCP network communications and XML data. We adopt P2P communication protocols (JXTA) to enable cross-firewall communications. Finally, the Sqlite3 embedded database is used to store data managed by the user MiddlewareEndPoint. This allows the middleware to persist data on disk by reducing the risk of data loss in case of crash or power-off. A schematic description of the user MiddlewareEndPoint is shown in figure 11.

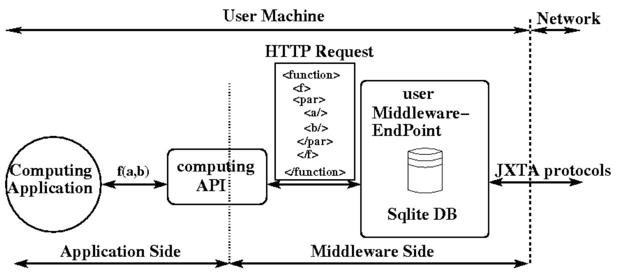

Figure 11. Computing API

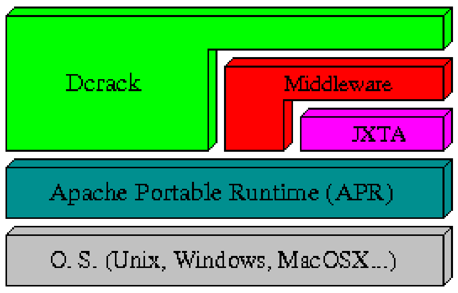
Dcrack
Middleware
JXTA
Apache Portable Runtime (APR)
O. S. (Unix, Windows, MacOSX...)

Figure 12. Software structure

B. *The* dcrack *application*

Dcrack 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:

task execution: each task is made of two components, the *worker* that acquires and processes information about the attack and the *server* that returns the 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 among them are shown in Figure 13. Task messages are sent from the Work component to the Server component of the application running on the parent node. Heart-beating messages are sent from the Beater component of the application running on the parent node to the Heart component running on the child nodes. Maintenance messages are exchanged between the controller component of the

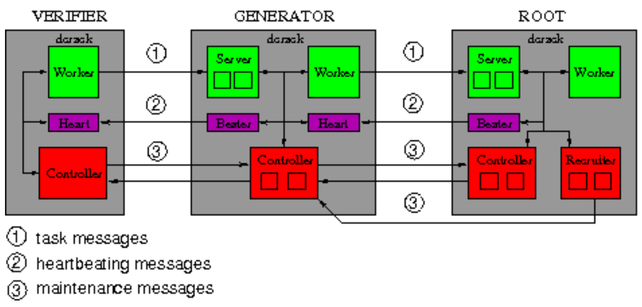

Figure 13. Communication scheme

application running on a child node and the corresponding component in the parent 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 among threads follows the *work-crew* model. Moreover, threads in charge of components that may require simultaneous communications (*i.e.,* the server, controller and recruiter components) generate a *service* thread to which the communication is demanded. Co-operation among component threads and service threads follows the *boss-worker* model.

C. Related Works

Medussa [1] is a scalable distributed cracking system supporting remote administration, multiple ciphers and multiple key generation methods. It uses a two-tier model, with a single server (*medussa*) and a number of nodes (*tentacle*). The server produces *keyspaces* and defines tasks scheduling on the basis of node performances and failures. In this way, it tries to improve the scalability and reliability of the system. The communication protocol is text-based and it is implemented on top of TCP. It is able to support binary transfers by the use of HTTP-like character encoding. Medussa supports multiple ciphers (*e.g.* Unix crypt, Single MD5, FreeBSD MD5 based crypt and Single SHA1) by means of OpenSSL's libcrypto. To match the cipher, Medussa allows the administrator to select the keyspace generator and supports three key generation methods: *binary* (all possible binary combinations), *dictionary* and *bruteforce* (ASCII based bruteforce keyspace generator). Medussa permits to define a hash and then associates a number of schedule runs with it. For example, a Unix crypt based hash can be first attacked using dictionary generators, and then a bruteforce attack. A successful run invalidates further ones. It is implemented as a classic Unix application, tested on Linux and Solaris operating systems, and the node-side can run also on Windows within a Unix compatibility environment. The main difference with our solution is that Medussa relies on a single server to manage scalability, reliability and keyspace production. Our three-tier architecture splits these tasks in two layers (the server and the generators) providing a more robust and reliable solution. Another key difference is the support Medussa provides for different cipher suites and key generation methods, whereas we support, at the moment, only one (GPG) cryptographic system and dictionary based key generation. As to the portability, we implemented our software on top of the APR library which guarantees portability on a number of Operating Systems. On the contrary, only the node-side (tentacle) of the Medussa project is portable on Windows.

Teracrack [2] exploits a time/space trade-off to take advantage of very large data storage capabilities, such as multi-terabyte disk systems, on password cracking using a dictionary based attack.

One goal of Teracrack was to pursue "world land-speed record" for password cracking, combining multi-teraflop computing, gigabit networks, and multi-terabyte file systems. With respect to other similar solutions, which deemed unfeasible to store generated password since they use "ordinary" hardware, Teracrack most characterizing feature is to exploit high performing hardware to store and manage a huge amount of resources required to produce and check pre-computed password. Our solution in completely different from Teracrack since we do not require a special kind of hardware, on the contrary, we exploit end-user desktop machines. We need to aggregate a high number of those machines to aggregate enough resources. To organize all the participating systems we defined a multi-layer architecture. This feature is totally absent in Teracrack. Moreover, we attack the GPG cryptographic system, that is more challenging than the Unix *crypt* function targeted by Teracrack. The only relevant similarity with our solution is the use of a dictionary based password generation method.

VI. 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, we used, in all tests, 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 set 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 IV-C.

VII. Conclusions and Future perspectives

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 VI, it is possible to introduce new generation rules and increase their complexity. Moreover, it is possible to use the same approach to attack other cryptosystems as described in the next section.

A. Application to Other Cryptosystems

Our distributed computing platform can be applied to perform dictionary attacks against any cryptosystem that adopts security mechanisms based on passwords/passphrases by using schemes of so called "password-based" cryptography. In the case of the OpenPGP format, passwords are secrets provided by users from which, by means of a hash function, a symmetric key is computed to encrypt permanently stored private keys. In many real applications, passwords are often used to generate symmetric keys that are directly involved in encryption of information to be protected. Since passwords typically do not have the required security properties to be used directly as cryptographic keys, a particular kind of functions, known in literature as "password key derivation functions" has been introduced following the definition used by RSA in the PKCS #5 standard (published as RFC 2898) [11]. In particular, to strengthen password security, two strategies are proposed [12]. The first is the introduction of the so called "salt", a sequence of n random bits that is prepended to a password, before to compute from it the hash value that will be used as key. In this way, the wideness of password space is multiplied by a 2^n factor. The second strategy consists in introducing key derivation techniques that are relatively expensive from the computational point of view, in order to increase cost and time required to perform an exhaustive search. A possible way to perform this task is to include an iteration count in the key derivation procedure, indicating how many times to iterate some underlying function from which keys are derived. Formally, a password key derivation function can be defined as follows:

$DK = PBKDF((P, S, f(.), c, l)$

where *i)* P is the password provided by the user; *ii)* S is the salt value; *iii)* $f(.)$ is the underlying function, that can be a hash (type $PKBDF1$) or a pseudorandom function (type $PKBDF2$); *iv)* c is the iteration count; *v)* l is the desired length of the derived key; *vi)* DK is the derived key;

In the attack strategy described in previous sections, the dictionary compilation phase and the passphrase generation phase are substantially independent from the cryptosystem under attack. Conversely, the passphrase verification phase is strictly dependent on verification process. As a consequence, in the architecture of the system, nodes placed at the first two levels (root and generators), are independent from the cryptosystem, whereas the verifier nodes must check all the generated passphrases according to the specific verification process of the password key derivation function adopted by the cryptosystem to be attacked. Moreover, that process depends on the cryptographic algorithm used for symmetric encryption and on the procedure performed by the cryptosystem in order to test the cryptographic key derived from the trial passphrase.

From the practical point of view, the main difficulty in extending the proposed methodology to other software cryptosystems is a consequence of the fact that many of them are not publicly documented with all necessary details and they are often closed source. In those cases, an expensive reverse engineering work is required, in order to understand how the cryptosystem involves the passphrase in the encryption/decryption operations.

In the field of disk encryption, a noteworthy exception to this trend is the open-source software Truecrypt [13].

The proposed attack methodology can be applied to this cryptographic software by implementing in verifier nodes the control procedure performed by the application when an encrypted volume is mounted.

Truecrypt uses a so called "header key" to encrypt/decrypt data in the volume header, that contains, among other data, the master key. In order to generate the header key, it uses a password key derivation function where the underlying function is a pseudorandom function (type $PKBDF2$), based on HMAC-SHA-512, HMAC-RIPEMD-160 or HMAC-Whirlpool algorithms, with a 512-bit salt and 1000 or 2000 iterations. Symmetric algorithms adopted for encryption/decryption operations are AES-256, Serpent or Twofish.

When mounting a Truecrypt volume, the header encryption key derived from the user password is used to attempt the decryption of the first 512 bytes of the volume. The attempt to decrypt is considered successful if the first 4 bytes of the decrypted data contain the ASCII string "TRUE", and if the CRC-32 checksum of the last 256 bytes of the decrypted data matches the value located at byte 8 of the decrypted data. If these controls fail, the verification process is repeated, but acts now on bytes 65536-66047. If the conditions are not met again, mounting is interrupted.

An attack to the Truecrypt system based on our distributed computing infrastructure may focus on this preliminary phase and we are going to study the details of the procedure adopted by Truecrypt to implement a cost/effective validation process of potential passphrases as we did for GnuPG. Finally, in the development of the new application we expect to exploit, where available, the outstanding computational capabilities of modern Graphic Processing Units (GPU) that equip most of recent personal computers [14].

References

[1] "Medussa," http://www.bastard.net/ kos/medussa/medussa.html.
[2] T. Perrine and D. Kowatch, "Teracrack: Password cracking using teraflop and petabyte resources," *San Diego Supercomputer Center Security Group Technical Report*, 2003. [Online]. Available: http://security.sdsc.edu/publications/teracrack.pdf
[3] "Access Data Distributed Network Attack (DNA)," http://www.accessdata.com/descryptionTool.html, 2008.
[4] "Rfc4880: Openpgp message formats," http://www.faqs.org/rfcs/rfc4880.html.
[5] "The gnu privacy guard (gnupg)," http://www.gnupg.org/. [Online]. Available: http://www.gnupg.org/
[6] T. E. Gamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Transactions on Information Theory*, vol. IT-31, no. 4, pp. 469–472, 1985.
[7] "Rfc3174: Us secure hash algorithm 1 (sha1)," http://www.faqs.org/rfcs/rfc3174.html.
[8] "Rfc2144: The cast-128 encryption algorithm," http://www.faqs.org/rfcs/rfc2144.html.
[9] "Apache portable runtime (apr)," http://apr.apache.org/. [Online]. Available: http://apr.apache.org
[10] M. Bernaschi and E. Gabrielli, "MIDIC: a middleware for Volunteer Computing," in *Proceedings of the IEEE Workshop on Dependable Application Support in Self-Organising Networks (DASSON'07).*, Edinburgh, UK, June 2007.
[11] B. Kaliski, "PKCS 5: Password-Based Cryptography Specification Version 2.0," RFC 2898, 2000.
[12] R. Morris and K. Thompson, "Password security: A case history," *Communications of the ACM*, vol. 22, no. 11, pp. 594–597, 1979.
[13] "Truecrypt," http://www.truecrypt.org, 2008.
[14] S. A. Manavski, "Cuda compatible gpu as an efficient hardware accellerator for aes cryptography," in *Proceedings of the 2007 IEEE International Conference on Signal Processing and Communications (ICSPC 2007)*, Dubai, United Arab Emirates, November 2007.