GASSATA, a Genetic Algorithm as an Alternative Tool for Security Audit Trails Analysis

Ludovic MÉ
SUPÉLEC
B.P. 28
35511 Cesson Sévigné Cedex
France
lme@supelec-rennes.fr

Abstract

Security audit efficiency is low because the security officer has to manage such a huge amount of data recorded in the audit trail, that the task is humanly quite impossible.

Therefore, our objective is to design an automatic tool to increase the security audit trail analysis efficiency. The tool, so called GASSATA, for Genetic Algorithm for Simplified Security Audit Trails Analysis, should be viewed as an additional tool in the set of tools which allow the security officer to keep a sharp eye on potential intrusions.

The main ideas on which our work is based are the following : (i) anomaly detection[1] (i.e. answering the question "Is the user's behavior normal according to the past?") is well treated by tools as NIDES, so we choose to investigate misuse detection (i.e. answering the question "does the user's behavior correspond to a known attack described as an attack scenario?"), (ii) we have to detect intrusions on heterogeneous networks on which the construction of a global time is impossible, so we eliminate the timing aspect of the attack scenarii (that is the reason why we qualify our analysis by the "simplified" adjective) which are given as sets of events generated by the attacks, (iii) our approach is pessimistic in the sense that we try to explain the data contained in the audit trail by the occurrence of one or more attack, (iv) this problem of explanation is NP-Complete, so we use an heuristic method, genetic algorithms, to solve it.

[1] The statistical approach mainly used to enforce anomaly detection leads to some problems: the choice of the parameters of the statistical model is tricky, the statistical model leads to a flow of alarms in the case of a noticeable systems environment modification and a user can slowly change his behavior in order to cheat the system.

Our presentation is organised as follows. Section 1 presents our view of the security audit trail analysis problem. In section 2 we show how to apply genetic algorithms to this problem. Section 3 discusses our experiments which exhibit fairly good results. Finally, section 4 concludes and proposes furthur work.

1 Our View of the Security Audit Trail Analysis Problem

Formally, our approach can be expressed by the following statement:

- let N_e be the number of type of audit events and N_a the number of potential known attacks.
- let AE be an $N_e \times N_a$ attacks-events matrix which gives the set of events generated by each attack. AE_{ij} is the number of audit events of type i generated by the scenario j ($AE_{ij} \geq 0$) (See Fig. 4 for an example of such a matrix).
- let R be a N_a-dimensional weight vector, where R_i ($R_i > 0$) is the weight associated with the attack i (R_i is proportional to the risk inherent in the attack scenario i).
- let O be a N_e-dimensional vector where O_i counts the number of events of type i present in the audit trail (O is called "observed audit vector").
- let H be a N_a-dimensional hypothesis vector, where $H_i = 1$ if the attack i is present according to the hypothesis and $H_i = 0$ otherwise (H describes a particular attack subset).

To explain the data contained in the audit trail (i.e. O) by the occurrence of one or more attack, we have to find the H vector which maximizes the $R \times H$ product (it's the pessimistic approach: finding H so that the risk is the greatest), subject to the constraint $(AE.H)_i \leq O_i$, $(1 \leq i \leq N_a)$ (see Fig. 1).

Finding the "right" H vector is reductible to the zero-one integer programming problem which is known to be NP-complete. The application of classical algorithms is therefore impossible is our case where N_a equals to several hundreds.

The heuristic approach that we have chosen to solve that NP-complete problem is the following: a hypothesis is made (e.g. among the set of possible attacks, attacks i, j and k are present in the trail), the realism of the hypothesis is evaluated and, according to this evaluation, an improved hypothesis is tried, until a solution is found.

In order to evaluate a hypothesis corresponding to a particular subset of present attack, we count the number of events of each type generated by all

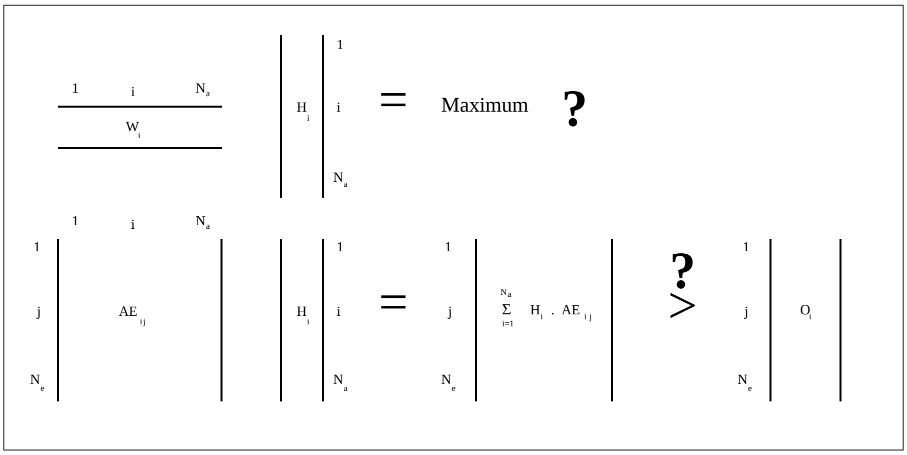

Figure 1: Our View of the Security Audit Trail Analysis Problem

the attacks of the hypothesis. If these numbers are less than or equal to the number of events recorded in the trail, then the hypothesis is realistic.

The last problem is to find an algorithm to derive a new hypothesis based on the past hypothesis: it is the role of the genetic algorithm.

2 Using Genetic Algorithms for Misuse Detection

Genetic algorithms (GA) are optimum search algorithms based on the mechanism of natural selection in a population. A population is a set of artificial creatures (individuals or chromosomes). These creatures are strings of length `l` coding a potential solution to the problem to be solved, most often with a binary alphabet. The size `L` of the population is constant. The population is nothing but a set of points in a search space. The population is randomly generated and then evolves: in every generation, a new set of artificial creatures is created using the fittest or pieces of the fittest individuals of the previous one. The fitness of each individual is simply the value of the function to be optimized (the fitness function) for the point corresponding to the individual. The iterative process of population creation is achieved by three basic genetic operators: selection (selects the fittest individuals), reproduction or crossover (promotes exploration of new regions of the search space by crossing over parts of individuals) and mutation (protects the population against an irrecoverable loss of information). The general structure of a GA is thus the following:

```
Random generation of the first generation
Repeat
    Individual Selection
    Reproduction
    Mutation
Until stop criteria is reached
```

Genetic operators are randomized ones but genetic algorithms are no simple random walks: they efficiently exploit historical information to speculate on new search points with expected improved performance.

Two sub-problems arise when applying GAs to a particular problem: (i) coding a solution for that problem with a string of bits and (ii) finding a fitness function to evaluate each individual of the population.

2.1 Coding a Solution with a Binary String.

An individual is a l length string coding a potential solution to the problem to be solved. In our case, the coding is straightforward: the length of an individual is N_a and each individual in the population corresponds to a particular H vector as defined in section 1.

2.2 The Fitness Function.

We have to search, among all the possible attack subsets, for the one which presents the greatest risk to the system. This results in the maximization of the product $R.H$. As GAs are optimum search algorithms, finding the maximum of a fitness function, we can easily conclude that in our case this function should be made equal to the product $R.H$. So we have:

$$Fitness = \sum_{i=1}^{N_a} R_i.I_i$$

where I is an individual.

This fitness function does not, however, take into account the constraint feature of our problem which implies that some hypotheses (i.e. some individuals) among the 2^{N_a} possible ones are not realistic. This is the case for some i type of events when $(AE.H)_i > O_i$. As a large number of individuals do not respect the constraint we decided to penalize them by reducing their fitness values. So we compute a penalty function (P) which increases as the realism of this individual decreases: let T_e be the number of types of events for which $(AE.H)_i > O_i$, the penalty function applied to such an H individual is then:

$$P = {T_e}^{p}$$

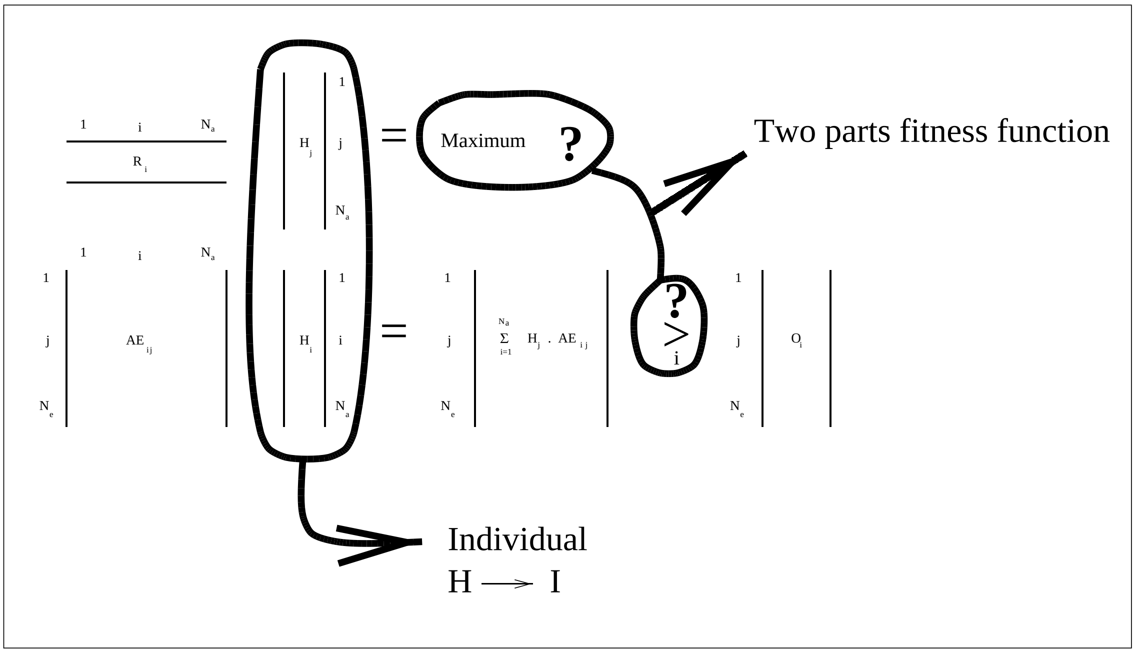

Figure 2: A Fitness Function for Using Genetic Algorithms on Misuse Detection

A quadratic penalty function (i.e. $p = 2$) allows a good discrimination among the individuals. The proposed fitness function is thus the following:

$$F(I_i) = \alpha + \left(\sum_{i=1}^{N_a} R_i.I_i \; - \beta.T_\epsilon{}^2\right)$$

The β parameter makes it possible to modify the slope of the penalty function and α sets a threshold making the fitness positive. If a negative fitness value is found, it is equaled to 0 and the corresponding individual cannot be selected. So the α parameter allows the elimination of too unrealistic hypotheses.

See figure 2.2 which illustrates the choices of the individuals and of the fitness function.

3 Experimental Results for Simulated Users and Attacks

3.1 Experimental Environment

The system AIX offers a security audit subsystem, allowing to generate various kinds of audit events which are recorded in a protected file. For our experiments, we use this audit subsystem.

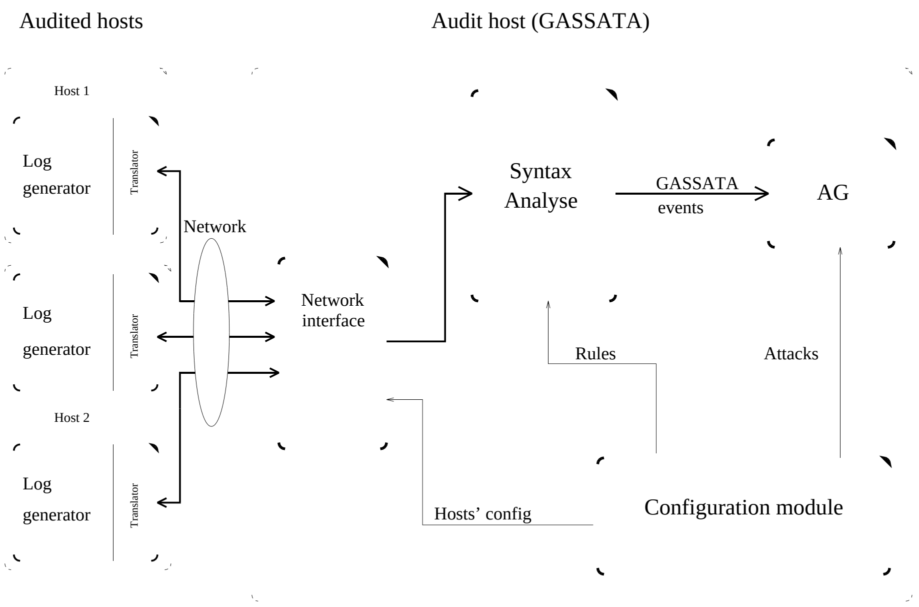

Figure 3: Architecture of GASSATA

Our prototype, GASSATA (see figure 3 which gives its architecture), finds the H vector which maximizes the $R.H$ product, subject to $(AE.H)_i \leq O_i$ $(1 \leq i \leq N_a)$. If the audit session is too long, this constraint is always enforced and GASSATA converges on the N_a-dimensional unit vector. To avoid this problem, the duration of the audit session should be chosen carefully. This is why we work on the basis of 30 minute audit sessions. We translate the audit trail into user-by-user audit vectors with a linear one-pass algorithm. (In a real environment, successive audit trails and audit vectors must be archived on tapes for possible future investigations.)

To perform realistic experiments on several user types, we defined the following 4 kinds of users: the inexperienced user, the novice developer, the professional developer and the UNIX intensive user. Each kind of user is defined by a sequence of commands which could be completed by this user over a 30 minute period. Each of these sequences is translated into an observed audit vector. We designed an Attacks-Events matrix including 24 different attacks. An example of an attack is given by the following commands[2], which allow the attacker to print any file (before step 2, the attacker has to make sure that there are jobs waiting in the print queue):

```
> touch f
> lpr -s f
```

[2] These commands have to be translated into AIX audit events which appear in the matrix. The length of 30 minutes is just an example. Nevertheless, the Attack-Events matrix corresponds to that length.

	Attaks																							
User_Login fail	3	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
User_Login (23h à 6h)	.	.	.	.	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
Short_Session	.	.	.	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
User_SU OK	.	3	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
User_SU fail	.	.	3	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
who, w, finger, f, ps...	.	3	.	.	.	.	.	.	8	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
more, pg, cat, vi, ...	.	.	.	.	.	5	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	5	.	.
ls OK	.	.	.	.	.	30	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
ls fail	.	.	.	.	.	.	5	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
df, hostname, uname	.	.	.	.	.	.	.	.	.	3	.	.	.	.	.	.	.	.	.	.	.	.	.	.
arp, netstat, ping	.	.	.	.	.	.	.	.	.	.	2	.	.	.	.	.	.	.	.	.	.	.	.	.
ypcat	.	.	.	.	.	.	.	.	.	.	.	3	.	.	.	.	.	.	.	.	.	.	.	.
lpr	.	.	.	.	.	.	.	.	.	.	.	.	.	.	10	1	.	.	.	.	.	.	.	.
rm, mv	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	.	.	.	.	.	.	.
ln	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	.	.	.	.	.	.	.
whoami, id	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	4
rexec, rlogin, rsh, ...	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	.	.	.
Proc_Execute	.	3	.	.	.	35	5	.	8	3	2	3	.	.	10	3	.	300	.	2	.	5	.	4
Proc_SetPri	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	100	.	.	.	.	.
File_Open fail	.	.	.	.	.	.	.	5	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
File_Open fail cp	.	.	.	.	.	.	.	.	.	.	.	.	.	10	.	.	.	.	.	.	.	.	.	.
File_Open .netrc	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	.	.	.
File_Read lpr	.	.	.	.	.	.	.	.	.	.	.	.	.	.	10	.	.	.	.	.	.	.	.	.
File_Read passwd, ...	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	5	.	.
File_Write passwd, ... fail	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.
File_Write cp OK	.	.	.	.	.	.	.	.	.	.	.	.	30	.	.	.	.	.	.	.	.	.	.	.
File_Unlink rm	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	50	.	.	.	.	.	.	.
File_Mode	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	3	.	.	.

Figure 4: An example of attacks matrix

```
> rm f
> ln -s /etc/security/passwd f
```

The number of events resulting from the 24 attacks is 28 (so we have here a 24×28 matrix, see figure 4). (We used also 200×28 matrix in order to appreciate the performances of GASSATA in terms of run times, see section 3.2 and figures 7 and 8.)

In order to appreciate the results given by GASSATA, we must know in advance the set of attacks really presents in the analysed audit trail. It is why attacks are simulated by including, in the observed audit vectors, events corresponding to one or more attacks. Moreover, we have to appreciate GASSATA's result when no attack is included in the trail.

Each experiment performed can be characterized by a 4-tuple (P_c, P_m, L, a) where P_c is the crossover probability, P_m is the mutation probability, L the population size, and a the number of attacks actually presents in the audit trail. For each value of the 4-tuple, we perform 10 runs (all the following results are averages over the 10 runs). From a genetic point of view, it is interesting to analyse the influence of each parameter. From a security point of view, the quality of the

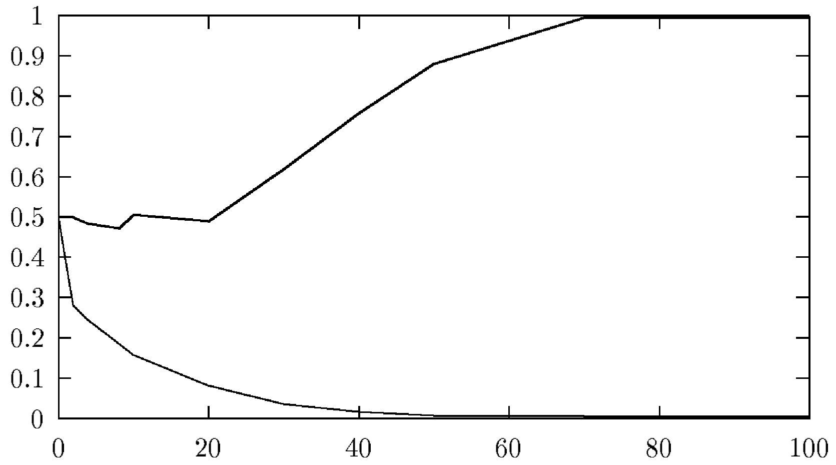

Figure 5: Evolution of T_p and T_a vs generation

detection is more important. We focus here on the security point of view and to appreciate the detection's quality, we define two ratios, T_p and T_a, as follows:

- T_p is the number of individuals in which bits corresponding to present attacks are 1 out of the total number L of individuals,
- T_a is the number of individuals in which bits corresponding to absent attacks are 1 out of the total number L of individuals.

We note T_{p_i} and T_{a_i} the values of T_p and T_a for the generation i. Thus, we have $T_{p_0} \simeq 0.5$ and $T_{a_0} \simeq 0.5$ (the initial population is randomly generated) and should have $T_{p_{final}} = 1$ and $T_{a_{final}} = 0$ (all the present attacks are detected and no absent attack is detected).

In order to appreciate the convergence of the population and the time needed for a detection, we compute, for each generation, the minimum, maximum and average whole population fitness values.

3.2 Results

Figure 5 shows the evolution of T_p and T_a versus the generation number (i.e. versus time). It shows that there is a good discrimination between present and absent attacks: the mean values of $T_{p_{100}}$ and $T_{a_{100}}$ are respectively 0.996 and 0.0044. We are close to the optimal values 1 and 0. The number of attacks actually present in the trail have no influence on this result.

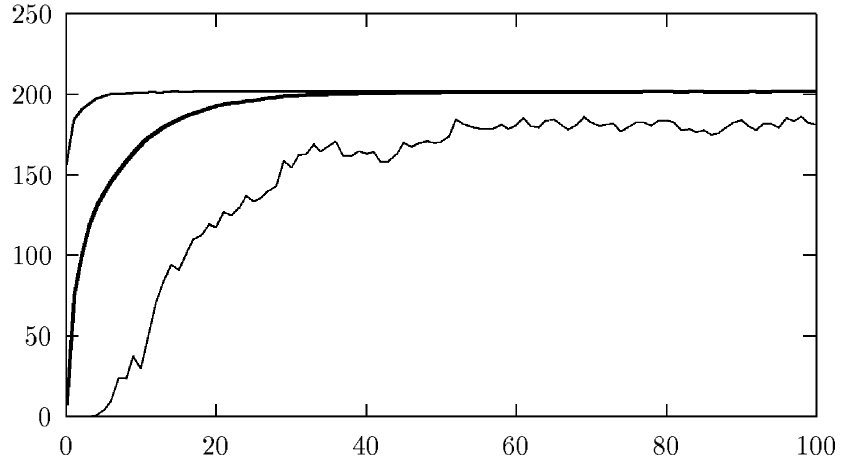

Figure 6: Average min, max and avg fitness for 10 runs ($\lambda = 500$, $P_c = 0.6$, $P_m = 0.002$ and $a = 2$) vs generation

Figure 6 shows that the maximum fitness value converges quickly on the optimum (after about 20 generations with a (24×28) Attacks-Events matrix. The remainder of the population follows and after 100 generations, the average fitness is about 99% of the maximum fitness. Once again, the number of attacks present in the trail have no influence on this result.

If the number of attacks coded in the Attacks-Events matrix grows, the final generation number has to increase to keep the detection's quality at the same level (i.e. $T_{p_{final}}$ and $T_{a_{final}}$ close to their optimal values). Figure 7 shows the evolution of the execution time when the number of attacks grows. When considering 200 attacks, GASSATA needs 10 minutes and 25 seconds to give the result of the analysis. Figure 8 compares the evolution of the number of possible solutions and the evolution of the execution time in seconds when the number of attacks coded in the matrix grows: the first grows exponentially whereas the second grows polynomially. When considering 24 attacks, the number of tried hypothesis out of the total number of possible solution is 0.003. With 100 attacks it becomes 5.9×10^{-26} and with 200 attacks 7.7×10^{-54}. It shows that genetic algorithms constitute a powerful heuristic method.

Finally, let us note that the duration of the audit session has no influence on the execution time because it only depends on the sizes of the two matrix H and AE which are constant.

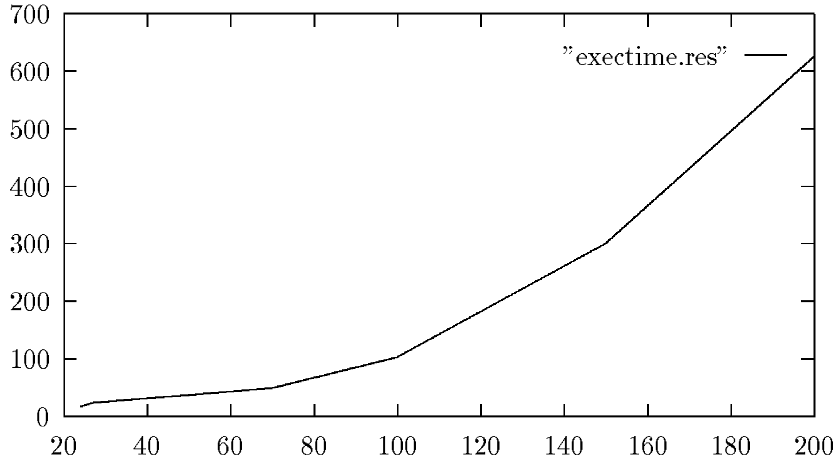

Figure 7: Execution time in seconds vs number of attacks in the matrix

4 Future Work

Some remaining problems (arising due to the use of predefined attack scenarios or to our simplified view of the problem) motivate future work :

- We are not able to take into account attacks characterized by event absence (e.g a programmer who does not use the cc compiler).
- By using a binary coding for the individuals, we cannot detect the multiple realization of a particular attack. As a consequence, we should try non-binary GAs.
- If the same event or group of events occurs in several attacks, an intruder realizing these attacks simultaneously does not duplicate this event or group of events. In that case, GASSATA fails to find the optimal H vector. We have no solution to that problem for the moment. This means that we only consider independent attacks.
- GASSATA does not precisely locate attacks in the audit trail. Just like statistical intrusion detection tools, it only gives a presumptive set of attacks present in a given audit session. The audit trail must be investigated later by the security officer to precisely locate the attacks.

Our experiments are pretty simple simulations. That first step of experimentation was necessary but now, our objective is to use GASSATA in a real

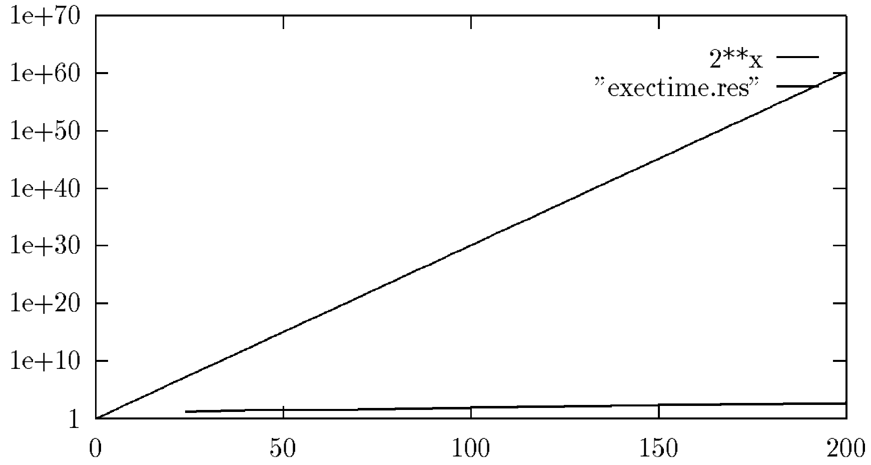

Figure 8: Number of possible solutions and execution time in seconds vs number of attacks in the matrix

environment, for a community of 100-200 real users generating actual audit data into which we will look for real potential attacks. That second step in experimentation will allow us to compare, on a practical basis, our tool with other ones.

See `http://www.supelec-rennes.fr/rennes/si/equipe/lme/these/these-lm.html` for more information on this work.