Abstract

PGP employs a decentralized trust architecture where there is no hierarchy of centralized agents who everybody trusts. Instead, everybody is treated equally in a "web of trust" approach. Everybody can send his public key to a PGP server which includes a binding between the public key and the key's owner. Everyone can also sign a certificate indicating the authenticity of another public key's binding to its owner, this certificate is also sent to the PGP server. However, it is more difficult to ascertain the authenticity of a public key, therefore it is very important for us to develop an algorithm to determine which public keys are trustworthy, and which ones are fake. In this paper, we will describe three existing algorithms for assigning trust to keys in the PGP network. Each of these algorithms havs its advantages and weaknesses. Overall, most existing algorithms do not provide a good way to detect fake public keys. We will introduce an evolutionary algorithm that not only determines which keys can be trusted, but also determines which keys are fake or malicious. The algorithm's runtime and accuracy are discussed.

1 Weaknesses of PGP

Every public key in PGP contains a name field, which indicates who the key belongs to. The main problem with PGP is that everyone is free to publish a key under any name to the PGP server, therefore we must establish a way to authenticate the binding between each key and its owner. As mentioned in the previous section, the "seeing is believing" protocal is usually used in PGP, where Alice will sign a certificate confirming a public key claiming to be Bob's indeed belongs to Bob only if she meets up with Bob personally. However, since there is not a centralized agent that oversees and regulates people signing certificates, it is possible for anyone to sign anyone else's certificate without actually meeting up. Therefore, it is possible for a malicious attackers to make a fake public key impersonating someone else, and make many other fake keys to confirm the authenticity of the former. This example illustrates that we cannot simply trust a key based on how many people have signed it. To make matters even worse, sometimes even an authentic key owner Alice might be forced by an attacker to sign a fake key, or she may simply make a mistake when signing. Thus it is desirable for us to develop an algorithm that can distinguish authentic and fake public keys even under

the situation where authentic users can occasionally make a mistake and sign a bad key.

It has to be mentioned that we cannot define a global value of trust for each key. To a person outside the PGP network, i.e. one who has not personally met up with anybody, there is no way for him to tell for certain which key are authentic and which ones are fake. As a contrived example, suppose an malicious user of PGP creates a fake copy of the real PGP network as follows: Whenever someone, A, joins the network, he creates a fake instance of that person A' at the same time. Whenever a certificate is signed between A and B, he makes a fake certificate using A' sign B'. To an outsider, the real and fake networks appear to be identitical, and there is no way to tell which is which. To differentiate the two networks, one must have personally met up with at least one of the users in the real network. For example, suppose C has personally met up and signed A's certificate, then he will know the network containing A, and not A', is the real one. Thus the lesson to be learned here is that trustworthiness can only be defined with respect to another user in the network. It makes no sense to say "A is trustworthy", instead we have to say "A can be trusted by B". In terms of our algorithm, to determine the trust value of A, we require as inputs both the network (including all the keys and certificates) and another person B, from whose perspective we are calculating A's trustworthiness.

2 Existing algorithms

2.1 Introducer Based PGP Trust Model

In this section we will discuss the standard algorithm that is used on most PGP networks to assign trustworthiness. The idea of the algorithm is the following:

Whenever Alice signs Bob's public key, she includes two values in the certificate:

1. How confident she is about the binding between the public key and its owner, Bob.

2. How confident she is about Bob being a trustworthy "introducer" to another trustworthy public-key certificate.

These are assigned separate values because Alice might believe that although the public key is indeed Bob's, but Bob, being a five-year-old, cannot

be reasonably expected to sign only other authentic user's keys, hence he is a bad introducer.

There are three levels of trust for the validity of a public key: undefined, marginal, and complete, as well as four levels of trust for a public key as an introducer: full, marginal, untrustworthy, and don't know [1]. It has to be mentioned that the level of trusts are not publically known, only the PGP server and the person who assigned these trust values know what they are. This is done to protect each PGP user's personal opinion on other people's trustworthiness.

To evaluate whether A should trust a public key B, the PGP server searches through the trusted paths from A to B. Since A does not know what trust value each person in the trusted path has assigned in their certificates, the trustworthiness of B is evaluated automatically by the PGP server, since only the server knows all the trust parameters. However, A can adjust certain security parameters (CompletesNeeded, MarginalNeeded) in the evaluation algorithm to reflect her skepticism level. To evaluate the level of trust of A on B, the PGP server computes two numbers:

CompletesCount: the number of paths from A to B where every certificate is completely trusted
MarginalCount: the number of paths from A to B where every certificate is at least marginally trusted.

Based on these numbers and the security parameters chosen by A, the PGP server outputs whether A should trust B completely, marginally, or not at all. For example, if CompletesCount $>$ CompletesNeeded, B will be marked as completely trust worthy [1].

There are a few issues with this approach. Firstly, people's opinions regarding the validity of a key and whether it is trustworthy as an introducer are subjective. Alice's final decision on whether to trust a key B depends on other people's opinions, and not hers, which is undesirable. Secondly, the CompleteCount and MarginalCount calculated by the PGP server increases as the number of keys in the PGP network increases. Ideally A should increase her security parameter depending on how big the network is, however, she may not know this information. Lastly, if an attacker can force a fake certificate from one of the people on A's completly trusted chain, then he can fork this path into many other completly trusted paths, which are all artificially constructed by the attacker, to arbitrarily increase the trust level

of any fake key according to A.

2.2 Disjoint Paths

In this algorithm, each public key in the PGP network is represented by a node. An edge connects A to B if A has signed B's public key. The algorithm proposes to base trust on the number of disjoint paths connecting A to B, where two paths are disjoint if they do not share a node. In particular, we want to bound the length of the paths from A to B by b. This problem is the Bounded Disjoint Path (BDP) problem [2]. We then assign k, the number of disjoint paths from A to B, as the level of trust of A in B.

Notice that this approach gets rid of the subjective levels of trust on the validity of each key as well as the level of trust on each key as an introducer. Every key and certificate are equal. This algorithm also fixes the last issue mentioned in the previous section, where an attacker can arbitrarily increase the trust value of a key if he can successfully fork the completely trusted chain from A. This is because all such paths would go through a single key, namely the compromised key in the trusted chain, so they will not be considered disjoint. Note that if there are k disjoint paths connecting A to B, then for B to be not authentic, at least k of the edges must be compromised. Therefore k is a reasonable measure of A's degree of confidence in B's authenticity. Unfortunately, the BDP problem is proven to be NP-hard. Therefore approximation algorithms must be used to evalute trust in a reasonable amount of time.

Although this algorithm is more robust against attacks than the PGP algorithm mentioned in the previous section, it is still not completely satisfactory, since:

1. The number of disjoint paths to any other key from A is bounded by the number of out-edges from A. Therefore, while the algorithm will provide a reasonable spectrum of trustworthiness if A has a large number of out-edges, it becomes much less indicative of trustworthiness if A only has, say, 3 out-edges, in which case all other keys will be assigned a trust value from 0 to 3.

2. By only counting the number of disjoint paths, the algorithm ignores other potentially valuable information contained in the topology of the graph. For example, there is only one disjoint path from A to B in Figure 1, yet we are less likely to trust B in the right scenario since if any one node in the graph is malicious, B cannot be trusted, whereas in the left scenario, there

is only one weak link, which is C.

3. It does not identify keys that are fake or malicious.

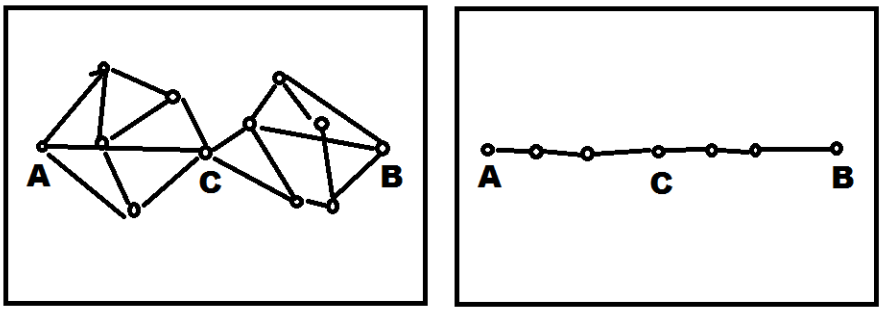

Figure 1:

3 Specification of Our Algorithm

The two existing algorithms described so far do not really detect malicious users, we hope to come up with an algorithm that can determine which users are trusted as well as which users are malicious. We hope to do this based solely on the topology of the network, instead of basing it on subjective opinions, like in the standard PGP trust algorithm.

Every public key in the PGP network will again be represented by a node. If key A signed key B, we represent this certificate with a directed edge. There is no additional information (such as subjective trust level) contained in the edges. In our algorithm, we wish to use a piece of information that is mostly ignored in other algorithms: name field. We will categorize the public keys into three types according to the nature of their name field:

Good: A public key whose name field correspond to its real owner.

Impersonated: A public key whose name field is identical to some good public key, but the key in fact belongs to an impersonator hoping to imitate someone else.

Madeup: A public key whose name field correspond to some made-up person.

After picking a source node, our algorithm should calculate the trust value of every other public key in the network according to the source node. In

particular, we wish to assign a trust value of 1 to good keys, and a trust value of 0 to impersonated and madeup keys.

Ideally, a good person should have only signed other good people's keys. Thus, good nodes should only have edges to other good nodes. Other than that, there is no limitation on who the bad nodes (impersonated and madeup) can sign.

Now, define a **Person set** to be all nodes whose name field correspond to the same person, say Obama.

Among the nodes in the Obama person set, there will be good public keys that indeed belong to Obama, as well as bad public keys that are by definition impersonators. We can "merge" the good nodes belonging to Obama into a single node, and simplify the problem by effectively making each person set to have at most 1 good keys. This is done by requiring that every key belonging to the real Obama should sign each other, creating a clique. Note that an impersonator may be able to sign Obama's real keys, but will never get signed in return. Thus, if every good user follows this rule when adding new keys to the network, the nodes in each person set naturally breaks up into sets of nodes belonging to different purported owners, and only one of these owners is real.

We can merge a group of nodes belonging to the same owner into one super node, which combines all the in-edges and out-edges connecting to nodes outside the group. This way, we have effectively enforced that every real person can have only 1 key. We thus arrive at the following rule in our model:

Person set constraint: Among every person set, there is at most one key with trust 1, the rest have trust 0.

Note that it is still possible for every key in a person set to be assigned 0, when Obama is not in the network, and all keys claiming to be him are malicious.

3.1 In a Perfect World

First, let's see how we can implement our algorithm in a world where no good key every signs a bad key (either impersonated or madeup). Given a directed graph, and a source node A. We wish to assign a 1 to every node that can be trusted by A, and 0 to every impersonated or madeup node.

step 1: A must be assigned 1 (it makes no sense for A to not trust her-

self). Since we are in a perfect world, every person A signs must be good, so we assign all children from node A a trust value of 1. Again, the children of A's children must also make perfect judgements and sign good keys. Thus, any node that can be reached from A are all assigned 1. This can be done with BFS.

step 2: Now we will make use of the name field data to identify malicious users. By the person set constraint, we can look through all the person sets belonging to people we've established to be good in step 1, and assign everyone else in these sets a trust value of 0.

step 3: Now that we've established a set of impersonators, we can expand the set of bad nodes by looking at which keys signed the bad nodes. Since good keys in our model never sign bad keys, whoever signed a bad key must be bad itself. I.e. the parent node of a bad node must be bad. We can again use BFS to trace through all the parents of bad nodes, and assign them 0.

step 4: the remaining nodes should be identified as inconclusive.

3.2 Imperfect World and Signing Violations

The algorithm described above will become useless if some good key signs a bad key either by mistake or by coercion. This is because the trust values assigned to each node will be ill-defined, for example, a node can be both the children of a good node, but also the parent of a bad node. Given a graph and an assignment of 1 and 0 to each node, define **signing violation** to be an edge going from a trust 1 node to a trust 0 node.

We expect the number of signing violations to be quite rare in a real network. Thus, an assignment that results in fewer number of signing violations will be more likely than another assignment that results in more. So, our algorithm will attempt to find an assignment of trust values that results in the least number of signing violations.

Our problem:

Given: a directed graph, a source node A which is assigned a value 1, sets of nodes belonging to the same person set

Objective: find an assignment of 0 and 1 on every node that results in a minimum number of signing violations (edge connecting 1 to 0), while satisfying the person set constraint (at most one node

can be assigned 1 within each person set).

4 Naive algorithm

The straight forward yet extremely inefficient solution to the problem is to try every single assignment of each node that is consistent with the person set constraint, and determine which one has the smallest number of signing violations. Clearly, this algorithm runs in exponential time, and is impractical. We suspect the problem is NP-hard, but cannot offer a proof.

We wrote up the naive algorithm anyways and ran it on a few test networks. Since it is infeasible to generate a large network by hand, we wrote a function to generate a random network with the following parameters:

1. nGood: number of good nodes
2. nImp: number of impersonated nodes
3. nMadeup: number of madeup nodes
4. nCertGood: Maximum number of certificates (out edges) from each good node
5. nCertBad: Maximum number of certificates (out edges) from each impersonated or madeup node

Here, each good node has a different name field (by definition), and each impersonated node randomly chooses from the list of good nodes to impersonate. Each madeup node has its own unique name field just like the good nodes. The number of certificates issued by any good node is a random integer between 1 and nCertGood; similarly, the number of certificates issued by any bad node is a random integer between 1 and nCertBad. We further require that good nodes can only sign other good nodes, whereas bad nodes can sign any node, good or bad.

Notice that a network generated as such does not contain a signing violation. We will introduce signing violation with another function Violate(network, nViolation):

Where nViolation indicates the number of signing violations we wish to introduce. Each signing violation is generated by randomly picking a good node and a bad node and have the good node sign the bad node.

We tested the naive algorithm with a randomly generated network of various sizes (nGood+nImp+nMadeup). We find that the run time of the algorithm is on the order of a few minutes for a size of 50, and becomes much

longer for anything above 50. Exactly what we expect for an exponential time algorithm.

5 Evolutionary Algorithm

The naive algorithm clearly does not work for large networks. We thus turn our attention to a randomized and approximate algorithm which will hopefully run faster. After exploring a few different ideas, we find that an evolutionary algorithm works the best. The basic idea of the evolutionary algorithm is as follows:

1. We randomly assign each node 1 or 0. Develop some metric to compute the "badness" score of this assignment. Repeat this N times, and select the M assignments with the lowest score.

2. From each of the M assignments, we produce N/M "offsprings". Each offspring has roughly the same assignment as its parent, except each assignment can be flipped randomly with a probability r.

3. We will again end up with N assignments, we pick out the best M assignments and repeat the procedure nGen times.

In our simulations, we used N = 100, M = 5, r = 0.01.

5.1 Measuring Badness

The most crucial component of the evolutionary algorithm is a good choice of a function that evaluates how bad a particular assignment is. The badness score should in principle be composed of two parts: signing violations, and person set inconsistencies.

We first tested the evolutionary algorithm with the most simple minded function:

$$\text{BadScore} = \#\text{ of signing violations} + \#\text{ of inconsistent person sets} \quad (1)$$

Where an inconsistent person set has more than one person being assigned a trust value of 1.

We observed that with this function, the algorithm often fails to find the optimal solution. In particular, the end result of the algorithm sometimes involves all nodes being assigned 1. This is because while there are only O(n) person groups (where n is the size of the network), there can be as much as O(nc) signing violations. Where c is the average number of certificates

issued by any node (for simplicity we will assume nCertGood = nCertBad). Therefore, the majority of the points in this function come from signing violations, and the person set inconsistency score is negligible. Note that if we assign everybody a trust of 1, we totally avoid signing violations while getting penalized only for person set inconsistencies, which is a small penalty. Thus, we should increase the value of person set inconsistency scores by a factor of O(c). In practice, we find c/4 to be a reasonable choice. so we modify our BadScore function to be:

$$\text{BadScore} = \#\text{ of signing violations} + \frac{c}{4} \times \#\text{ of inconsistent person sets} \quad (2)$$

We find that while this function works reasonably well for small networks, it fails on larger ones. The standard failure mode is for the algorithm to assign 0 to every node except the source node (which is force to be 1). The reason for this is also easily understood. If we assign 0 to everything we can totally avoid person set inconsistencies, while also avoiding most of signing violations except the ones going from the source node to its children.

Once the assignment is in a state where everything except the source is assigned 0, it is very difficult to get out of through evolution. As shown in Figure 2, where s is the source node and c is one of its children. For c to flip back to 1, we'd avoid one signing violation from s to c, but introduce 4 more from c to its children. Therefore, the shortsightedness of our evolutionary algorithm will eliminate this change from the next generation.

To fix this problem, we make edges going from s to its children worth more points than other edges. It is found that a reasonable choice is c/2.

However, there are still problems with this method on even larger graphs. Where the failure output of the algorithm is to assign the source and its children 1, and everything else 0. The reason is similar, consider a grandchildren g of s, which is assigned zero along with everything else except s and its children. To flip g's assignment to 1, we eliminate 1 signing violation but introduce many more. Thus, we need to assign a higher weight for edges connecting s's children to grand children.

We can extend this train of thought and assign each edge a different weight in calculating the signing violationg score based on how far it is from s. In practice, we can stop this after just a few generations until more than half of the nodes are covered. So, in our final BadScore function, the signing violation score is modified to introduce higher weights for edges in the vincinity of the source node.

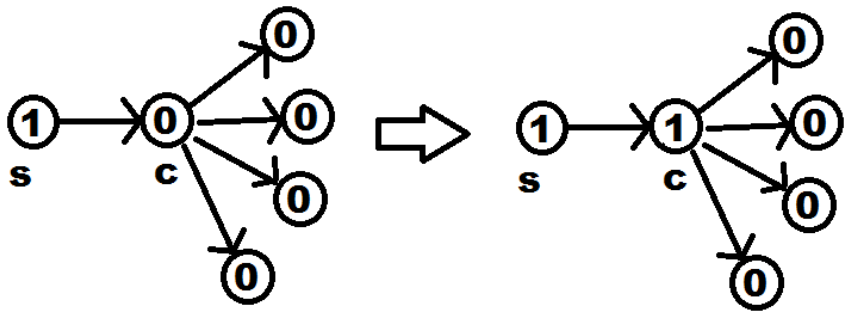

Figure 2:

6 Results

We tested our evolutionary algorithm in terms of speed, and resistance against attacks. In particular we tried two different types of attacks. First, we we gradually increaase the number of violations in our graph and see if our algorithm can still accuratly assign trust values. This corresponds to attackers being able to force more bad keys to be signed by good people. Second, we increase the number of bad nodes in the graph, this corresponds to an attack where malicious users flood the network hoping that perhaps at least some of the bad nodes will be falsely identified as trustworthy.

6.1 Runtime

In this section, we run our algorithm on randomly generated networks of increasing size. The run time of each generation is roughly proportional to O(nc) where n is the size of the network and c is the average number of certificates signed by each node. Firstly, we have demonstrated this algorithm to work for $n > 300$, recall that with the naive algorithm we only went up to $n = 50$. We ran a series of tests with $n = 100, 150, 200, 250$ respectively. Figure 3 shows the BadScore decreasing as the generation number. In all cases, the badness score converged within 200 generations.

To visualize what how the algorithm does. We plot the best trust assignment (1 for white, 0 for black) in each generation as a horizontal line of pixel, and each successive generation is drawn underneath the previous generation's assignmment. The plot is shown in Figure 4

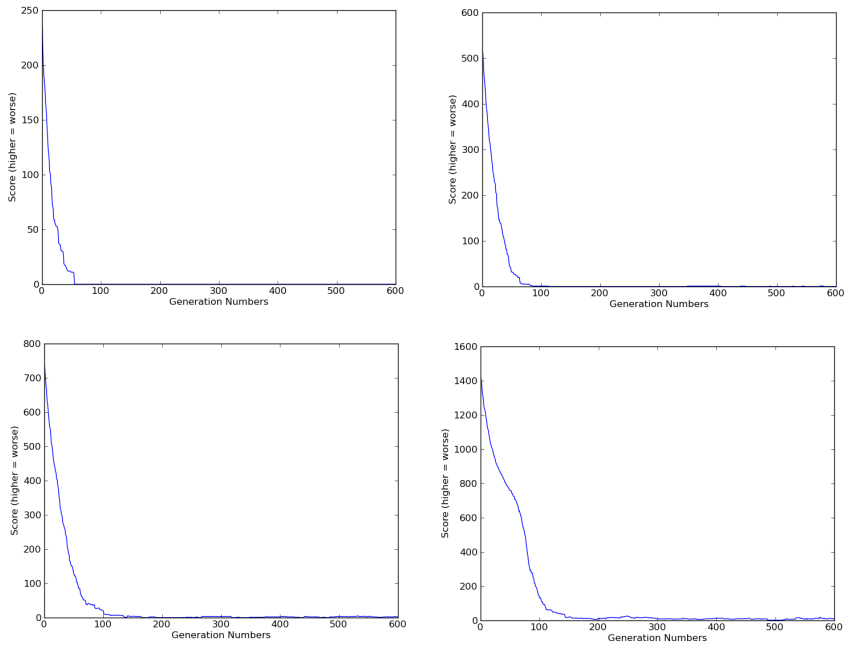

Figure 3: Convergence time for n = 100 (top left), 150 (top right), 200 (bottom left), 250 (bottom right)

The first 100 pixels correspond to good nodes, the next 100 correspond to impersonated nodes, and the last 50 correspond to madeup nodes. The algorithm quickly assigns the correct trust values to the good and impersonated node, and it mostly works for the madeup nodes as well.

The reason why some madeup nodes are assigned a value of 1 is that, since the network is generated randomly, some madeup nodes can be assigned either 0 or 1 without changing the BadScore, these nodes correspond to the inconclusive nodes in the naive algorithm.

In conclusion, our evolutionary algorithm clearly converges very quickly, and it is clearly sub-exponential.

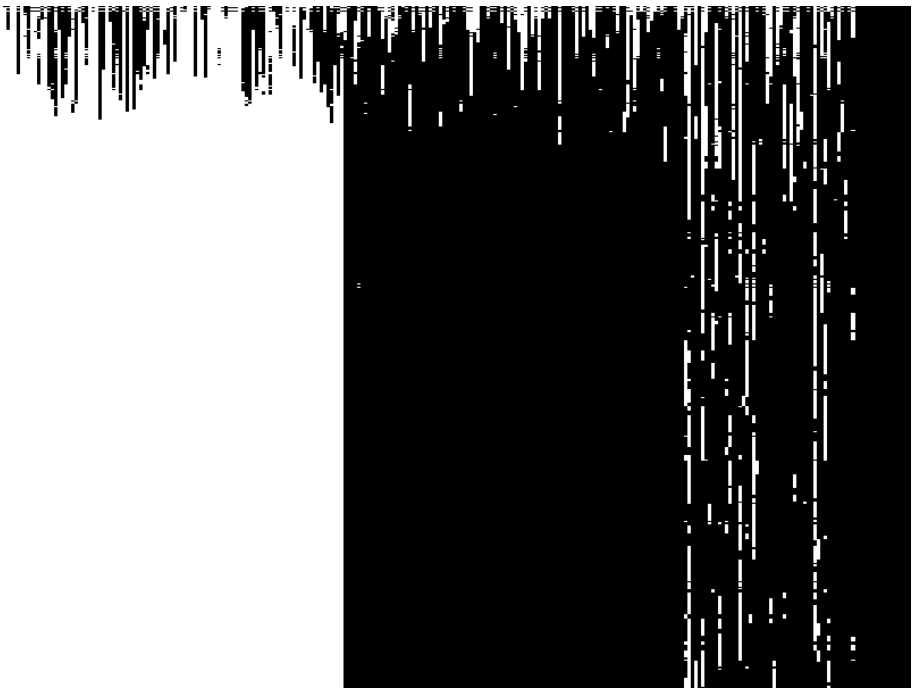

Figure 4:

7 Resistance Against Large Number of Violations

Next, we gradually increase the number of signing violations in our graph. We worked with a graph of 125 nodes (50 good, 50 impersonated, 25 madeup). We set the max certificate signed by each user to be 20. Therefore the expected total number of certificates signed by the good users is $50 * 20/2 = 500$. We gradually increase the number of signing violations from 0 to 300. And plot the accuracy rate of our algorithm. Where a completely accurate assignment assigns 1 to every good node and 0 to every impersonated or madeup node. The result is ploted in 5

The graph shows that the accuracy seems to either be really close to 1, or it is around 0.6. This is because when our algorithm fails, it sometimes assigns every node 0, as discussed before. When this happens, the accuracy is appproximately (nImp+nMadeup)/n = 75/125 = 0.6. Assigning each edge a higher weight if it is closer to the source can mitigate this problem, but it

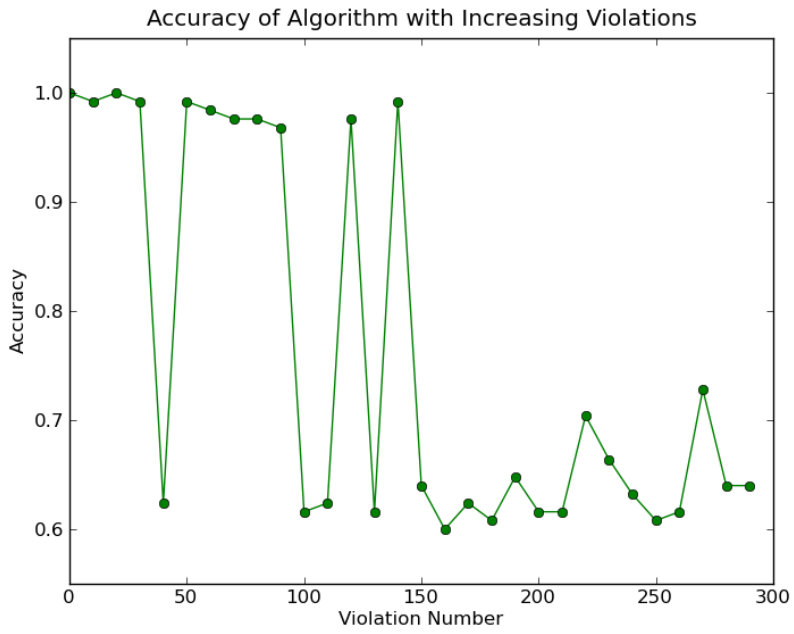

Figure 5: Accuracy against signing violations

cannot totally avoid it. Since we are running a randomized algorithm on a randomized network, sometimes our algorithm will fail and yield an accuracy of 0.6. Still, from the graph, we can see that the success probability of our algorithm greatly decreases when the number of signing violation is above 125.

Since there are a total of 500 good certificates, this correspond to approximately 20% of all certificates signed by good nodes are to bad nodes. This is quite robust, because it means to significantly decrease the accuracy of our algorithm, attackers need to force at least 20% of all certificates from good nodes to be bad. It has to be mentioned that this ratio will likely vary depending on the ratio of (good, impersonated, madeup) nodes, as well as how many certificates each node signs.

8 Resistance Against Large Number of Bad Public Keys

In this scenario we fix the number of good certificates to be 50, we fix the total number of violation to be 50, and we increase (nImp, nMadeup) from (40,40) to (120,120). The accuracy of the algorithm is shown in Figure 6

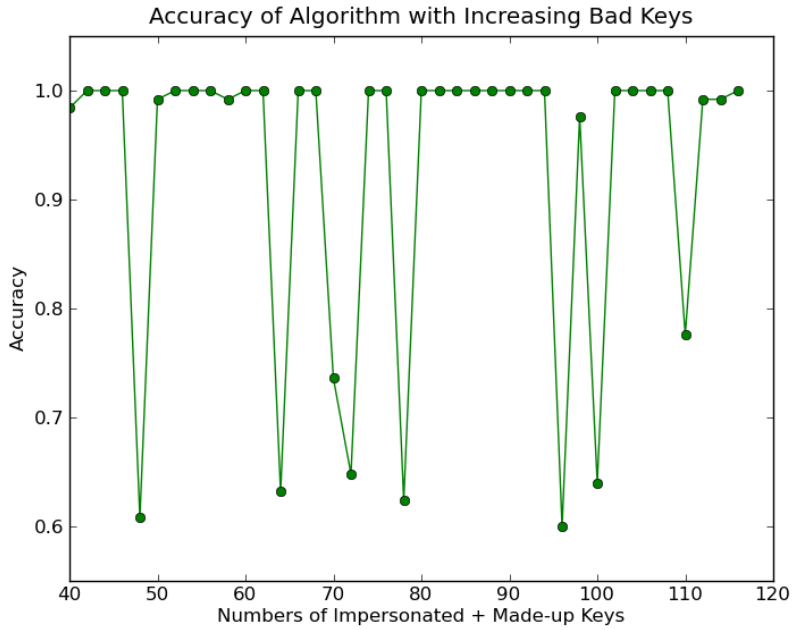

Figure 6: Accuracy against number of bad keys

The graph shows that the accuracy of the algorithm does not change dramatically as we increase the number of bad keys up to about 80% of the total number of keys. This shows that our algorithm is resistant against such attacks.

9 Weaknesses of Our Algorithm

A key difference between our algorithm and other existing algorithms is that our algorithm uses information contained in the name field. With this, we can

categorize users into good, impersonated, and madeup. The name field helps us define person sets, and the constraints satisfied by each person set gives us information regarding which keys are bad. However, if we completely remove impersonated keys from the network, leaving only good keys and madeup keys, then there will never be any person group inconsistencies. Hence all our BadScore will come from signing violations, which can be minimized to zero if we just assign everyone in the network to 1. In this scenario, our algorithm no longer works since it will just assign everyone to be trustworthy.

Therefore our algorithm must be used in conjuction with other existing algorithms. For example, our algorithm can be used on a network to first identify all the bad nodes, eliminate them from the graph, and then rank how much we can trust the remaining good nodes with either the standard PGP algorithm or the disjoint path algorithm described in the beginning.

Also, since our algorithm must assign either a 1 or 0 to each node, for nodes that should be categorized as inconclusive, our algorithm will assign them a value anyways. For example, if a node is completely isolated, it will have a 50-50 chance of either being trustworthy or not, instead of being categorized as inconclusive. Therefore we might also run an algorithm on the network first to pickout all the nodes that we can infer are inconclusive.

10 Code

Our code can be found on the web at: *https://github.com/haoqili/PGP_Trust_Ranking*.

11 Conclusion

In the first part of our project, we made it possible to sign PGP keys on the Android phone with a QR code scanner. In the second part of our project, we created a evolutionary algorithm to assign a trust value values to all nodes, given a graph and a source node as inputs. Our algorithm runs in sub-exponential time. Our algorithm is also resistant to signinng violations with the ability to tolerate about 20% of all certificates signed by good keys that are violations. Finally, our algorithm is resistant to flooding of bad keys that at least 80% of all keys can be either impersonated or madeup without signifcant decrease in the accuracy of the algorithm.

References

[1] A. Abdul-Pahman. The PGP Trust Model. *EDI-Forum: the Journal of Electronic Commerce*, April 1997.

[2] M. K. Reiter and S. G. Stubblebine. Resilient Authentication using Path Independence. *IEEE Transactions on Computers* vol. 47, no. 12, pp. 1351-1362, December 1998.