Trust without Truth

Michel Deriaz
University of Geneva, Switzerland
Michel.Deriaz@cui.unige.ch

Abstract. Can we trust without any reliable truth information? Most trust architectures work in a similar way: a trustor makes some observations, rates the trustee, and makes recommendations to his friends. When he faces a new case, he checks his trust table and uses recommendations given by trustworthy friends to decide whether he will undertake a given action. But what if the observations that are used to update the trust tables are wrong? How to deal with what we call the "uncertainty of the truth"? This paper presents how people that publish and remove virtual tags are able to create trust relations between them. A simulator as well as a concrete and widely deployed application have been used to validate our model. We observed good and encouraging results in general, but also some weaknesses, brought out through specific scenarios.

1 Introduction

Spatial messaging, also called digital graffiti, air graffiti, or splash messaging, allows a user to publish a geo-referenced note so that any other user that attends the same place can get the message. For example, let us consider the community of the Mt-Blanc mountain guides. The members would like to inform their colleagues about dangers in specific places or about vacancies in refuges. One guide can publish a geo-referenced message that informs about a high risk of avalanches, and any other guide that attends the same place will get the warning, and comment it if necessary. It is a kind of blog, in which editors and readers share the same physical place.

There are many reasons to believe that spatial messaging will become a wide spread concept in a nearby future. Today, people use the connection capabilities of their mobile phone mostly in one way, to download information. But in the same way that people passed from television to Internet, the next generation of user will probably become "active" and publish information. If we remember how fast the computer power and the communication capabilities of these devices improve, and the fact that there are today more modern mobile phones (with Internet connection) than desktop computers in the world, we can easily paint a glorious future for mobile technology. This assertion can be confirmed by the growing interest for location awareness. The success of Google Map Mobile [1], a service that allows you to download maps on your mobile phone as well as POIs (Points Of Interest) wherever you are, is an indicator of this growing interest. And Google Map Mobile is not alone. There are more and more applications or Internet services for mobile users that provide maps and other information related to your current position.

There are already some implementations of the spatial messaging concept, but experiences realized with volunteers showed that there is only little interest in posting notes. To our view, the main reason is that there is currently no trust mechanism which informs about the reliability of the messages, thus preventing any serious application. In our Mt-Blanc mountain guides example, even if the security aspects will ensure that the posted messages are really posted by the mentioned author, that no modifications of the original text can be made afterwards, and that the service is available for everyone that is authorized, you still need a trust mechanism to know how reputable the author is.

This paper proposes a generic model to handle the trust component in spatial messaging. We validated it through a simulator and through a widely deployed application called FoxyTag, which allows a driver to publish virtual tags near traffic radars in order to warn the other drivers.

2 A new model is required

Lots of work has already been done in the trust context, and the obvious question that arises is why not just using well-known trust models? The answer is simply that it will not work. Indeed, traditional trust models are mainly designed with file sharing or auctions applications in mind. In this case, people are rating each other and when user *A* wants to download a file (or buy an item) from user *B*, he questions the system in order to determine how trustworthy user *B* is. Currently, commercial systems (like e-Bay) are using very basic centralized systems, and the academics are suggesting solutions to transform such systems into peer-to-peer architectures. But spatial messaging is noticeably different from file sharing or auctioning. First of all, we want to take care about the context. For example time is important. Imagine that you see during summer time a tag that warns about a high risk of avalanches. Even if there is no snow anymore, it does not mean necessarily that the author was lying; it can also mean that the tag has been written six month ago. Second, we believe that trust cannot only be applied to users. The tags themselves have to maintain information so that a user can compute how reliable it is to him.

In traditional computational trust, we usually agree over a set of axioms and hypothesis. For instance, the "truth" is a notion that is common to all. A corrupted file is seen as corrupted by everybody. In spatial messaging however, the truth is context dependant. The truth becomes a subjective and temporal notion. Something that is true for one user is not necessarily true for the others. Something that is true at a certain time is not necessarily true later. We call this new notion the "uncertainty of the truth". If user *A* posts a tag saying "Dangerous path", user *B* only knows that user *A* finds this path dangerous. But *A* is perhaps just a tourist and the path is in no way dangerous for user *B*, how can be a confirmed mountain guide. Or this path was maybe dangerous because of the snow, which has melted away by the time.

To our view, trust is not only a tool that can be used to exclude malevolent users from a given system. Trust is also a way of creating relationships between users that behave in a similar way. Like in real life, each user has his own definition of what the truth is. The aim is therefore to create trust relationships between people that share the same definition.

3 Related work

We already tackled the time component in a paper that has been published in the PST'06 proceedings [2]. In the survey, we wrote that several authors are aware about the difficulty to take the time into account, but no one proposed a trust model that gracefully solved the problem, or at least it was not directly applicable to spatial messaging. Dimmock [3], who realized the risk module in the EU-funded SECURE project [4], concluded in its PhD thesis that "one area that the framework does not currently address in great detail is the notion of time." Guha [5] built a generic trust engine allowing people to rate the content and the former ratings. He recognized however that in case of highly dynamic systems (like in spatial messaging where tags can appear and disappear very quickly), "Understanding the time-dependent properties of such systems and exploiting these properties is another potentially useful line of inquiry." Most existing trust metrics update their trust values only after a specific action, like a direct interaction or the reception of a recommendation. The few trust engines that take the time component into consideration simply suggest that the trust value decreases with the time. Mezzetti's trust metric [6] consists in multiplying the trust value at time *t* by a constant between 0 and 1. We proposed in [7] a similar model that also takes into consideration the dispersion of the outcomes. In Bayesian-based trust metrics [8, 9], the trust value converges to its initial value over time. All these models work in situations where the changes occur slowly, but are challenged in short-lived cases.

Our former time-patterned trust metric, called TIPP GC (TIme-Patterned Probabilistic Global Centralized), was used in a collaborative application allowing to signal speed cameras on mobile phones. A full description of the trust engine and the application can be found at [2]. Even if we brought some novelties about the way we updated the trust values, we still used a "traditional" way to store them, i.e. the number of positive outcomes *P* and the number of negative outcomes *N*. The trust value equaled $P / (N + P)$. And under a certain trust value, the malevolent users were simply excluded from the system. The problem with this kind of metrics is that it is difficult to decrease the trust value of a user that behaved correctly for a long time. We suggest therefore, to be closer to the human way of handling trust, that any trust value must decrease quickly in case of bad behavior. An honest user that becomes malevolent must not be able to use its long term good reputation to subvert the system.

4 Our model

4.1 Overview

Spatial messaging is not a new concept [10, 11], but existing systems do not have a trust mechanism, thus preventing any serious application. We can of course build a trust engine for each application, but it is like reinventing the wheel each time. Worse, the trust engine is the more complicated part.

Our solution to this problem consisted in building a framework that provides, among other things, a generic trust engine. So that it becomes very easy to build new applications using trusted virtual tags. Our framework, called GeoVTag, provides an API that eases the development of new applications using virtual tags.

To facilitate further comparisons, we introduce here a second scenario that is quite different from the mountain guides one. It is FoxyTag, a collaborative system to signal speed cameras on mobile phones. The idea consists in posting virtual tags close to radars in order to warn other drivers. These users will then get an alarm when they are closer than 15 seconds to a critical point, and a red point locating the radar appears on their screen. A driver signals a radar by pressing the key "1" of his mobile phone and signals that a radar disappeared (he gets an alarm but he does not see any speed camera) by pressing "0".

Creating a single trust engine that fits all the different applications is a difficult task. One reason is because the way we compute a trust value differs from one situation to another. There are different classes of trust engines. For instance we have situations where changes are unpredictable, like in the FoxyTag scenario where a radar can appear or disappear at any time. What if you get an alarm but you do not see any speed camera? You do not know if the former driver was a spammer (and then you need to decrease its trust value) or if the radar simply disappeared. But there are also situations where changes are more predictable. In the mountain guides scenario, if someone warns about a danger of avalanches, he can easily put a deadline to his tag, thus avoiding disturbing with an outdated tag a user attending the same place six months later.

It is clear that we compute the trust differently when the tags are meant to change often than in situations where the tags are meant to be stable. In the FoxyTag scenario, we could handle differently fixed radars and mobile ones. A mobile speed camera that disappears after a few hours is a "normal" situation. But a fixed speed camera that disappears is an unusual situation, especially if other neighboring radars disappear as well.

The GeoVTag framework provides a generic trust engine that can be easily extended. Updates in the trust table are made according to the behaviors of the users, and each of this update can be redefined and configured via rules and parameters. Roughly speaking, the designer of a new application will have to code "how much a specific behavior in a specific context costs in terms of trust value". He will therefore only have to code behaviors directly related to its application, leaving the framework doing all the job of maintaining and managing the trust information.

The main idea of our trust engine is to remember only important or recent information, like it is done in human communities. Tags and users keep a history of their last or important transactions. To know whether a tag must be shown to the user, the trust engine checks the *n* last reviews done by trustworthy users. A user is trustworthy if its global trust value, computed has a mix of the trustor's opinion (based on former direct interactions) and the opinions of the trustor's friends (who ask their own friends, and so on until a certain level), is above a certain threshold. A trustor calls friend every user with who he has a good trust relationship, or better said, each user with a good local trust value. That was how to get a tag. When a user rates a tag, he updates the trust values of the author and the former reviewers according to rules and parameters that depend on the application. In certain cases, a review can be done on both directions. For instance an author can update the trust value of every reviewer that gives a positive rating, since they seem to share the same opinion about the tag. However, these "reverse reviewings" must be configured with greatest care, to avoid that a malevolent user rates automatically and positively all the tags he crosses, in order to use its growing trust value to subvert the system.

4.2 A vTag in GeoVTag

A vTag is a virtual tag. It contains the following fields:

- **ID**. A unique identifier for this tag.
- **Author**. The ID of the author. This field, which is an integer, equals -1 when an author decides to revoke its own tag.
- **Position**. The geographical position of the tag. Each tag is attached to a given position, expressed in latitude and longitude.
- **Creation time**. The time when the tag has been created.
- **Deadline**. After the deadline, the tag is removed.
- **RD (Request to delete time)**. To avoid malevolent acts, it is not possible for a user to directly remove a tag. Instead, when certain conditions are met (for instance several users that rated the tag negatively), a "request to delete" is made to the tag. Its value is the time the request is made, and external rules define when the tag should be definitively removed.
- **Content**. The content of the tag. It is the application that decides how to structure the content. For instance an application could decide that the content is always an URL, and that all the tags are coded in HTML.
- **Reviewers**. A user can agree or disagree with the content of a tag. A tag contains a reviewers list that is sorted in an inverse chronological order. Each review contains the current time, the ID of the reviewer, the rating, and possibly some content (same format as the content written by the author).

These are the minimum fields required by the trust engine. An application designer can however add his own ones, like for instance the area where the tag is visible, under what condition it is visible...

4.3 A user in GeoVTag

A user is composed of an ID and a trust table. After an interaction with user *B*, user *A* updates the local trust value of *B* and places *B* on top of its list, so that there are sorted in an inverse chronological order. Each trust value is simply an integer in the range [t_{min}, t_{max}] so that $t_{min} < 0 < t_{max}$. GeoVTag allows specifying rules to describe how a trust value must be changed according to a given situation. A typical case is to have a linear way to increase a value (for instance adding *n* when you agree with a tag) and an exponential way to decrease a value (for instance multiplying by *m* a negative trust value). When $-t_{min}$ is much bigger than t_{max} (for instance t_{min} =-70 and t_{max} =5), we imitate the human way of handling trust: Trust takes time to be built, we forgive some small misbehaviors (exponential functions moves slowly at the beginning), but when we loose trust in someone (one big disappointment or lots of small disappointments) then it becomes very difficult to rebuild a good trust relationship. We avoid that malevolent users switch between good behaviors (in order to increase their trust value) and bad behaviors (in order to subvert the system).

It is important that our system forgives small mistakes in cases where the truth is unknown. We recall here the driver that gets an alarm about a speed camera that does not exist anymore. He will disagree with the author of the tag as well as with all the people that agreed. He will therefore decrease their trust values since they are perhaps spammers. But, most likely, the radar simply disappeared in the meantime and they are not spammers. Our model is built to forget easily such mistakes, as long as they do not happen too often, but to decrease quickly the trust values of malevolent users.

The global trust value of a user is relative and is computed by the following function:

$$global\_trust = q * myOpinion + (1-q) * friendsOpinions\ , \quad q=[0..1]$$

It is a recursive function where *myOpinion* is the local trust value and *friendsOpinions* is the average opinion of the *n* first friends (where *local_trust* >= 0). These friends apply the same function, so they return a mix between their own opinion and the average opinion of their own friends. And so on until we reached the specified depth. This way of processing is fast (all the values are centralized) and gives a good idea of the global reputation of a user. Typically, if we choose *n*=10 (number of friends) and a depth level of 3, then we have already the opinion of $10^0 + 10^1 + 10^2 + 10^3 = 1111$ reliable people, with more importance given to close friends. The more *q* is big, the more the user gives importance to it own value. In situations where people are susceptible of doing mistakes, this value is usually quite small.

4.4 The GeoVTag framework

The GeoVTag framework facilitates the development of applications using virtual tags. A simplified view of the framework can be seen in figure 1.

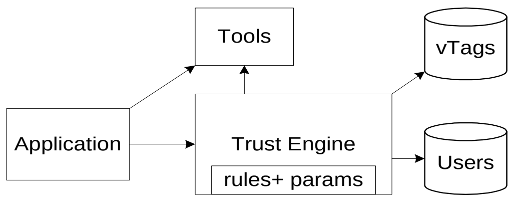

Fig. 1 GeoVTag framework

The Tools box is used by the trust engine and can also be accessed by the application. It contains mostly geographical related tools, like methods allowing conversions or methods handling tags of different formats.

All accesses to the two databases (vTags and Users) are done via the trust engine. The way the trust values are updated is defined via the rules and the parameters. In short, an application designer will have to configure these rules (in practice he will extend the trust engine class and rewrite the methods that code each specific behavior), set the parameters, and then write its application.

The trust engine can be accessed via three main primitives:

- **setTag**. This primitive simply creates a new tag. No trust mechanism is used.
- **getTags**. Returns a list of tags. The requester specifies which filter he wants to apply to the result. For instance, a user can ask to get all the tags in a certain radius, with updated trust values for the author and the reviewers, and let the application decide what to do. But he can also ask to get only the tags that are above a certain trust level and ignore the others. Or he can apply a personal filter and not use the trust mechanism at all, like asking all the tags that are authored or reviewed by Alice.
- **reviewTag**. Reviewing a tag means to rate it, optionally to add a comment, and then update the trust tables of the reviewer, the author and the former reviewers. The way the trust tables are updated is defined through the rules and the parameters. The framework splits all the behaviors so that the application developer can simply write the rules according to the needs of its application.

5 Validation process

We chose a speed camera tagging application to validate our trust engine. The first reason is because the question is quite complex. As we saw previously, radars can appear and disappear at any time, and it is not always possible to know if a wrong alarm is due to spammers or if it is actually the radar that just disappeared. To our view, the speed camera application is a "top" problem, or a problem that deals with all the possible cases. If our trust engine works for speed camera tagging, it should also work for other applications. The second reason is that it was very easy to find volunteers to test our system, since they could save their money while increasing the road safety. We set up a simulator that allowed us to test different scenarios (spammers, users that try to delete all the tags...) as well as a widely deployed application used to confirm the results of the simulator.

5.1 The simulator

Our simulator randomly positions speed cameras on a road and runs the user's cars according to given scenario parameters. An additional user, whose behavior can also be completely specified, logs its observations and returns the number of true positives (alarm: yes, camera: yes), false positives (alarm: yes, camera: no), true negatives (alarm: no, camera: no) and false negatives (alarm: no, camera: yes).
We model our road as a single way on a highway. Exits are numbered between 1 and *n*. Between two exits there is only one speed camera, numbered between 1 and *n*-1. So the camera c1 is between exits e1 and e2, the camera c2 is between exits e2 and e3, and so on. Figure 2 shows a road model.

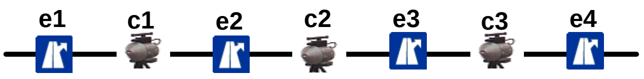

Fig. 2 The road model

This model seems to be very simplistic. It is however sufficient to validate our trust metrics. Of course, we do not take into account some contextual information, like shadow areas (tunnels, urban canyons...) or what happens when the user posts a message concerning the opposite direction. These are more technical issues that need to be validated in the field and it is what we actually did with a real device in a real car. Since we can define the behavior of every user (where they enter and exit, how reliable they are by signaling speed cameras...) as well as the behavior of each speed camera (frequency of turning on, for how long...), we can precisely define which user drives in which area and how many speed cameras he is meant to cross on average. Our simulator accepts an input file that looks like this:

```
cam;1-4;8;15,10  // about three times a day, for 15 minutes, 10 minutes pause
cam;5-5;24;2,0   // about once a day, for 2 minutes, no pause
cam;5-5;240;3,30 // about once every 10 days, for 3 minutes, 30 minutes pause
usr;1-10;1-5;24;95;90  // once a day, 95% true positive, 95% true negative
usr;1-1;3-5;240;80;75  // once every 10 days, 80% true positive, 75% true negative
usr;11-15;1-10;1;10;10 // every hour, 10% true positive, 10% true negative (hacker!)
usr;11-11;1-10;0;20;25 // every minute, 20% true positive, 25% true negative (hacker!)
col;5-7;1-11;6;10;100  // 4 times a day, 10% true positive, 100% true negative
spm;20-23;1-10;1       // every hour
scn;100;2;run(24);pas(1,10);act(1,10,50,60)
scn;10;4;run(2400);pas(3,5);run(1);act(1,10,100,100);run(2);act(1,10,100,100)
```

- In the first line, "cam;1-4;8;15,10" means that cameras 1 to 4 have one chance out of 8 to become active within an hour, and when one becomes active then it stays active for 15 minutes. After it stays inactive (paused) for at least 10 minutes. Note that these cameras will on average become active less than 3 times a day, since they cannot switch to active while there are already active or paused. Precisely, these cameras will become active every 8+(15+10)/60 = 8.42 hours.
- The next two lines define two different behaviors for camera 5.
- In the fourth line, "usr;1-10;1-5;24;95;90" means that users 1 to 10 entry the highway at 1 and exits it at 5, that they run once a day and that they vote 95% of the time correctly when they signal the presence of a speed camera, and 90% of the time correctly when they cancel a camera.
- In the collusion line, "col;5-7;1-11;6;10;100", we deduce that users 5 to 7 are colluding by entering all at the same time on entry 1, exiting on exit 11, and voting (all similarly) about all 6 hours with 10% of true positives and 100% of true negatives.
- In the spam line, "spm;20-23;1-10;1", we deduce that users 20 to 23 spam by entering all at the same time on entry 1, exiting on exit 10, and voting 1 about every hour at every speed camera place.
- The first scenario, "scn;100;2;run(24);pas(1,10);act(1,10,50,60)" contains 100 big loops and 2 small loops. The scenario itself will be executed twice, then the trust engine is initialized, and then we re-execute the scenario twice. And so on (100 times).
- run(t) means that the system will run for t hours (simulation time). Each minute, the go method of each camera and each user is called, allowing them to act according to their specified behaviors.
- pas(e1, e2) means that our test user will passively drive once from exit e1 to exit e2. Passively means that he does not vote. His observations are logged and printed.
- act(e1, e2, tp, tn) means that our test user will actively drive once from exit e1 to exit e2 and has tp (True Positive) chances (in %) to vote correctly if he sees a speed camera, and tn (True Negative) chances (in %) to vote correctly when he tries to cancel a speed camera that does not exist (anymore). His observations are logged and printed.
- Everything after a // is a comment and is ignored by the simulator.

5.2 Real life evaluation: FoxyTag

The simulator allows us to test the trust models, but how to be sure that our simulator acts in a way that is close to reality? To answer this question, we tested our model with FoxyTag [12], a collaborative system to signal radars on mobile phones. FoxyTag motivates neither speeding nor any other risky behavior, but allows the driver to concentrate on the road instead of having is eyes fixed on the speedometer, by fear of being flashed. We observe that drivers tend to brake suddenly when they see a radar (even if they are not speeding), which can provoke traffic jams or even accidents. FoxyTag signals in advance the presence of speed cameras, so that the driver has enough time to check its speed and adapt it if necessarily. A more technical description of this application can be found at [13].

5.3 Rules and parameters for the speed camera application

Each new user has an initial trust value equal to 0. A user is meant to send "1" if he sees a radar, or "0" if he gets an alarm but does not see any radar. If the application gets a "1" and there is no neighboring camera (less than 150 meters), it is considered as a creation of a new tag. If there is a neighboring camera, this "1" is considered as a positive rating for the existing one. A "0" is therefore considered as a negative rating. The main parameters are the following:

- **Minimum trust value**. t_{min} = -70. A malevolent user can have a trust value as low as -70. This is to make sure that a malevolent user cannot easily regain a good reputation in order to subvert the system a second time.
- **Maximum trust value**. t_{max} = 5. It is not possible to have a trust value higher than 5. The reason is that a user can suddenly change its behavior and become malevolent. This means that even if a user behaved correctly for many years, he will not be able to use his past good behavior to subvert the system.
- **Size of the history**. It is the number of ratings that a tag keeps in memory. A new rating will automatically erase the oldest one. If a user already rated a tag, the old rating is deleted and the new one is put on top of the list. We chose 10 for this value, so we keep only recent information. This value could seem small, but is perfectly adapted to an environment where changes can happen very suddenly.
- **Number of contacts**. This is the number of contacts that each user keeps, or the size of its trust table. Each time the user modifies the trust value of another user, the later takes the first place in the trust table. If a new user appears and there is no place in the trust table, the last one (the one that did not get any rating for the longest time) is removed. We chose 1000 for this number.
- **Weight of user's opinion**. We saw previously that the reputation of a user is computed as a mix of the user's own value (local trust value) and the one given by its friends. This parameter defines the weight to give to the user's opinion. We chose 0.2, meaning that we take 20% of the user's own opinion and 80% of his friends' opinions.
- **Number of levels**. When we need the global trust value for a given user, we ask our friends, who ask their own friends, and so on up to a certain level. We chose 2, meaning that we get the opinion of our friends and the friends of our friends.
- **Request to delete threshold**. The number of successive users that must deny the tag (voting 0) in order to make a request to delete. We chose 2 for this value.

The rules are described below:

- **Vote 1 for 1**. Confirming a tag. The 8 first people that confirm a tag increase by 5 the author's trust value and the author does the same with these reviewers.

- **Vote 1 for 0**. The previous reviewer denied the tag, but it seems the radar still exists. Its trust value is decreased by 3. It is not reasonable to decrease by more than 3 since it can simply be a misuse (mixing up buttons...) of the application. And since there must be at least 2 successive reviewers that deny the tag before a request to delete is made, this error will not harm the quality of the system.
- **Vote 1 for 00**. The two previous reviewers denied the tag, but it seems the radar still exists. This time the chance of being a misuse is reduced and this pattern could be considered as two malevolent users trying to remove the tag. Their trust values are updated like $t' = t * 1.5 - 5$, so that a misuse can be easily forgiven but if this behavior is repeated then the trust value falls quickly.
- **Vote 0 for 1**. The previous reviewer confirmed the existence of the speed camera but it seems that there is no radar anymore. It can reflect a normal situation (the radar simply disappeared), so the trust value should not be decreased too much. But it can also be the result of a spammer attack. Since a spammers attack is less dangerous than a deniers' one, we observed that decreasing the trust value by 1 in this case is not too penalizing for honest users, and still sufficient to exclude spammers in a reasonable delay.
- **Vote 0 for 0**. This case happens when a second user denies a tag. The two users increase mutually their trust value by 5.
- **Request to delete**. This rule defines when a tag that got a request to delete order (in our case after two successive disapprovals) should be removed. We decided to keep it for the same amount of time than elapsed between the creation time and the request to delete order, but for at least 6 hours and at maximum 50 days. A long term tag (for instance a fixed speed camera) will therefore need more time to be deleted. The minimum of 6 hours avoids that two malevolent users scan the network and delete all the tags as soon as they appear without being penalized by the trust engine.

These rules motivate the users' participation. Posting or confirming a tag increases trust relationships. We could think that it is not a good idea to deny a tag when the radar disappeared. It is true that in such a case we decrease (-1) the trust value of the previous reviewer who was probably an honest user. But on the other hand, we will build a bidirectional trust relationship with the second user that will deny the tag, and the increase of the trust values (2 times +5) compensates generously the former loss.

6 Results

In addition to our new trust model, we ran also the simulator on two very easy trust engines that have been used for comparison. The first is called "Test" and simply adds a tag when a user sends a "1" and removes it when a "0" is sent. The second one is called "Basic" and works as follow:

- If a user sees and mentions a new camera, a new tag is created. The default value of its counter equals 0.

- If a user sees and mentions an existing camera (one that was signalized by a tag), the corresponding tag counter is set to 1.
- If a user gets an alarm about a camera that does not exist anymore and mentions it, the counter of the corresponding tag is decreased by 1.
- A tag whose counter reaches -1 is deleted.

The main idea behind these rules is that if a user signals by mistake a new speed camera, then the next user can alone cancel the message, but if a second driver confirms the existence of a speed camera, then we need two people to remove the tag.

Now let's see the scenarios and the results. Scenario 1 tests our trust engine when malevolent users try to remove all the tags.

Scenario 1

cam;1-10;0;9999999;0
usr;1-100;1-11;24;100;100
usr;101-105;1-11;1;0;100
scn;100;100;run(24);act(1,11,100,100)

	tp - yy	fp - yn	tn - nn	fn - ny
Test	43470	0	0	56530
Basic	59450	0	0	40550
SC	99781	0	0	219

We have 10 radars that are always turned on, a hundred users that behave always correctly and five users that systematically try to cancel all speed cameras they cross. Each hacker runs on average 24 times more often than an honest user. In the results table we compare the Test, the Basic and the SC (we call our new trust engine SpeCam) trust engines. We used also the following abbreviations: "tp - yy" means true positives (alarm: yes, camera: yes), "fp - yn" means false positives (alarm: yes, camera: no), "tn - nn" means true negatives (alarm: no, camera: no) and "fn - ny" means false negatives (alarm: no, camera: yes).

With the Test trust engine, we see that there are more false negatives (alarm: no, camera: yes) than true positives (alarm: yes, camera: yes). This is normal since the malevolent users are driving more than the honest ones. But our SpeCam trust engine eliminates quite well these malevolent users, since less than 0.22% (219 / 99781) speed cameras where not tagged.

Scenario 2

cam;1-10;9999999;0;0
usr;1-100;1-11;24;100;100
spm;101-105;1-11;1
scn;100;100;run(24);act(1,11,100,100)

	tp - yy	fp - yn	tn - nn	fn - ny
Test	0	20550	79450	0
Basic	0	36110	63890	0
SC	0	240	99760	0

Scenario 2 tests how the trust engine reacts against a spammers attack. This time the cameras are always turned off and the malevolent users vote "1" for each radar position. Again we observe a significant improvement with our new trust engine.

Scenario 3

cam;1-10;48;360;720
usr;1-100;1-11;24;100;100
scn;100;100;run(24);act(1,11,100,100)

	tp - yy	fp - yn	tn - nn	fn - ny
Test	8736	346	90572	346
Basic	8734	688	90245	333
SC	8692	674	90304	330

In scenario 3 we have 10 radars that are turned on every 66 hours (48 + (360 + 720) / 60) for 6 hours, and 100 users that vote always correctly. We expected therefore similar results than for the Basic trust engine, which seems to be the case.

Scenario 4

cam;1-10;48;360;720
usr;1-100;1-11;24;95;95
scn;100;100;run(24);act(1,11,95,95)

	tp - yy	fp - yn	tn - nn	fn - ny
Test	8356	350	90510	784
Basic	8751	750	90090	409
SC	8710	836	90056	398

In scenario 4 the users are voting incorrectly 5% of the time. This figure is clearly overrated (according to the tests realized with FoxyTag where this number is less than 1% in practice), but it let us to prove that our trust engine is tolerant with unintentional incorrect votes made by honest users.

Scenario 5

cam;1-10;48;360;720
usr;1-100;1-11;24;100;100
usr;101-105;1-11;1;0;100
scn;100;100;run(24);act(1,11,100,100)

	tp - yy	fp - yn	tn - nn	fn - ny
Test	3885	58	90901	5156
Basic	5123	115	90873	3889
SC	8726	820	90047	407

In scenario 5 we added 5 deniers that try to remove all the tags they cross. The honest users are behaving correctly 100% of the time. We have clearly more false positives than for the Basic trust engine. This is normal since the deniers removed all the tags, whether there is a camera or not. If we compare the results with the ones from scenario 4, we see that our trust engine eliminates efficiently deniers, since the number of false positives and false negatives are similar.

Scenario 6

cam;1-10;48;360;720
usr;1-100;1-11;24;95;95
usr;101-105;1-11;1;0;100
scn;100;100;run(24);act(1,11,95,95)

	tp - yy	fp - yn	tn - nn	fn - ny
Test	3653	67	90927	5353
Basic	5051	129	90717	4103
SC	8623	920	90020	437

In scenario 6 the users vote incorrectly 5% of the time. Unfortunately, we observe that the number of false negatives and false positives increase a little bit (compared to scenario 5). It seems that 5% of incorrect votes is a critical limit for this scenario.

Scenario 7

cam;1-10;48;360;720
usr;1-100;1-11;24;100;100
spm;101-105;1-11;1
scn;100;100;run(24);act(1,11,100,100)

	tp - yy	fp - yn	tn - nn	fn - ny
Test	8656	18348	72768	228
Basic	8910	32978	57937	175
SC	8777	1591	89223	409

In scenario 7 we replaced the deniers by a spammer team, who votes "1" at every radar position. The other users are voting correctly 100% of the time. Even if the number of false negatives is correct (compared to scenario 3), we observe a high number of false positives. We first thought of a weakness in our trust engine, but further investigations concluded that it is actually the simulator that presents a weakness. The problem is that the positions of the radars are always the same (which is not the case in reality), and that sometimes, by chance, a spammer really signal a new speed camera, which generously increases its trust value. In reality this would not be a problem, since signaling randomly a real speed camera at the right place is almost impossible.

Scenario 8

cam;1-10;48;360;720
usr;1-100;1-11;24;95;95
spm;101-105;1-11;1
scn;100;100;run(24);act(1,11,95,95)

	tp - yy	fp - yn	tn - nn	fn - ny
Test	8440	19048	71941	571
Basic	8867	35156	55769	208
SC	8652	1761	89176	411

In scenario 8 the honest users are voting incorrectly 5% of the time. We face the same weakness as in scenario 7. However, to scope with this problem, we tried to remove from the system all the users where the mean trust value (average of the local trust values of all the users) falls under -2. We got then similar figures than in scenario 3, meaning that these "bad" values are mainly due to the simulator and not to the trust engine.

7 Conclusion

We set up a trust engine that deals with what we call the "uncertainty of the truth" or a situation where a trustor rates a trustee according to an observation that not necessarily reflects the truth. Our trust engine is generic and can be adapted through rules and parameters to any application using virtual tags. We chose the topic of speed camera tagging since it is a complex problem in terms of uncertainty (speed cameras can appear and disappear in a very unpredictable way) and since it was easy to find volunteers to test our application.

The results presented in this paper where computed by our simulator, and some of them where compared with data collected by FoxyTag (a widespread application using our trust engine) in order to make sure that our simulator behaves in a way close to reality. We observed that our trust engine excludes malevolent users but "forgives" small mistakes (due to the "uncertainty of the truth") and infrequent misuses (incorrect votes due a mix of the buttons) done by honest ones.

The main weakness we discovered in our work was directly related to the simulator. Since the positions of the speed cameras where always the same, spammers could by chance signal real radars and then have their trust value generously increased. The second weakness was due to our trust engine and precisely with scenario 6. We saw that in case of a heavy attack, the honest users had to do less than 5% of incorrect ratings in order to keep the system reliable. In practice this is not really a problem since we observed that real people using the application do less than 1% of incorrect votes.

The next step in our study will be to use the deadline parameter of our tags. In the speed camera case, we will be able to differentiate mobile radars from fixed ones. We expect then an improvement in the presented figures, since we will be able to set more precise rules.

References

[1] Website: http://www.google.com/gmm/

[2] M. Deriaz and J.-M. Seigneur, "Trust and Security in Spatial Messaging: FoxyTag, the Speed Camera Case Study", in Proceedings of the 3rd International Conference on Privacy, Security and Trust, ACM, 2006.

[3] N. Dimmock, "Using trust and risk for access control in Global Computing", PhD thesis, University of Cambridge, 2005.

[4] Website: http://secure.dsg.cs.tcd.ie/

[5] R. Guha, "Open Rating Systems", 1st Workshop on Friend of a Friend, Social Networking and the Semantic Web, 2004.

[6] N. Mezzetti, "A Socially Inspired Reputation Model", in Proceedings of EuroPKI, 2004.

[7] M. Deriaz, "What is Trust? My Own Point of View", ASG technical report, 2006.

[8] S. Buchegger and J.-Y. Le Boudec, "A Robust Reputation System for P2P and Mobile Ad-hoc Networks", in Proceedings of the Second Workshop on the Economics of Peer-to-Peer Systems, 2004.

[9] D. Quercia, S. Hailes, and L. Capra, "B-trust: Bayesian Trust Framework for Pervasive Computing”, in Proceedings of the 4th International Conference on Trust Management (iTrust), LNCS, Springer, 2006.

[10] J. Burrell and G.K. Gay, "E-graffiti: evaluating real-world use of a context-aware system", in Interacting with Computers, 14 (4) p. 301-312.

[11] P. Persson, F. Espinoza, P. Fagerberg, A. Sandin, and R. Cöster, "GeoNotes: A Location-based Information System for Public Spaces", in Höök, Benyon, and Munro (eds.), Readings in Social Navigation of Information Space, Springer (2000).

[12] Website: http://www.foxytag.com

[13] M. Deriaz and J.-M. Seigneur, "FoxyTag", ASG technical report, 2006.