Anonymous Identity and Trust for Peer-to-Peer Networks

Tom Murphy VII, Amit K. Manjhi

April 29, 2002

Abstract

In this paper, we present a new way of establishing independently-verifiable identities, based on the notion of computationally expensive key generation. We then describe a fully decentralized framework where these identities can be used to assign blame and to construct auditable blacklists of cheaters.

1 Introduction

With the proliferation of broadband in the home, peer-to-peer networks are becoming more and more popular. Though they are primarily used for file sharing (because their high degree of anonymity and low amount of liability), tremendous possibilities exist for their use for other purposes. (Several examples are given throughout this paper.)

Unfortunately, true peer-to-peer networks face a number of problems. One of the most daunting is the difficulty in policing a network without centralized authority. One of the major factors that makes this hard is the anonymity of the participants. Even if one user determines that another user is up to no good, he has almost no chance of successfully sharing this information with others—there's no way for him to uniquely identify the troublemaker, and no reason for anybody to believe him, anyway.

This paper is organized as follows. First, we present a new way of establishing independently-verifiable identities, based on the notion of computationally expensive key generation. We then discuss different models of trust, including one well-suited to peer-to-peer networks. We then describe a fully decentralized framework where these identities can be used to assign blame and to construct auditable blacklists of cheaters. Finally, we propose several other applications that could benefit from our scheme, and discuss some related work.

2 Identity

The notion of identity is subtle, and the word has many possible meanings. One might consider identity as something fixed, or varying over time, or related to autonomy, or even something metaphysical like a soul or consciousness. For our purposes, we consider identity simply to be a way of aggregating responsibility for past actions.

Identities are not necessarily in one-to-one correspondence with individuals. Bruce Wayne maintains two identities, that of "Bruce" and that of "Batman", in order to protect his estate from his arch nemeses. Corporations in the United States are given special identities to financially protect the owners and shareholders.

Since identities aggregate blame (especially) for actions, there is an implicit element of trust when sharing identities. If Bruce lets his butler Alfred use his credit card to order things online, he does so because he is willing to share responsibility for Alfred's actions. When Alfred creates a new username and password every time he posts irrelevant nonsense to the Gotham Gazette online message board, he does so to keep the administrators from being able to successfully ban him. That is, he doesn't want his past actions to be reflected in his new identities, so he simply generates a new identity for each infraction.

When identities are assigned by a central authority, such as in the form of Social Security Numbers or Corporate Tax IDs, it is difficult to amass a collection of disposable identities. However, in a decentralized system without trusted authorities, this becomes a more difficult problem. Some systems use information such as IP or MAC addresses, but this is an unsatisfying solution because of factors such as Dynamic IPs, Network Address Translation, IP spoofing, and shared or public computers.

In a decentralized system, we can't rely on a trusted authority to limit the number of identities that Alfred can generate. Therefore, we must rely on *intrinsic* properties of the identity. Furthermore, we need to be able to independently verify identities, so that anyone can check that someone is who he claims he is. To do this, we combine two notions from cryptography: public/private keys, and one-way functions.

2.1 Computationally Expensive Identity Generation

Our system has the following properties:

1. Allows the generation of an identity with a specified *strength*
2. Associates a token with the identity that allows for the easy independent verification of its strength. Others know an identity by this token
3. Allows for signing of messages, certifying that they came from a particular identity
4. Allows the strength of an identity to grow over time

We're able to achieve most of these goals by simply using a standard public key cryptosystem such as RSA[4]. We take the identity to be the public/private key pair, and use the public key as the identifying token. By identifying a user only by his key, we preserve his "real world" anonymity while still being able to address him precisely on the network. [1]

However, we want to prevent Alfred from generating lots of keys. We do so by imposing an additional requirement on identifying tokens. Such a token must consist of a public key and an arbitrary bit string, which we call salt. Suppose the function `last` (s, n) returns the last n bits of string s. Then, the *strength* of such a token is the maximum b such that

$$\texttt{last}(\texttt{hash}(salt, pubkey), b) = \texttt{last}(pubkey, b)$$

That is, we require that the hash of the salt and the key collide with the key on the last b bits. Assuming that producing partial collisions for our hash function is difficult, generating a salt that yields a high key strength will be computationally expensive.

An identity token's strength can be independently verified by simply computing the hash and checking that the bits collide. This computation is simple and fast compared to the time it took to generate the salt. A user may choose to not accept a token that does not have at least a certain minimum strength, making it arbitrarily difficult for Alfred to generate many identities. We expect that the appropriate acceptable strength level will be determined through social means—a balancing of the annoyance of generating a key with the amount of throw-away identities that users are willing to tolerate. We guess that a dozen or so CPU-hours on a modern machine would be in the right ballpark.

Now, since it takes significant resources to generate and maintain keys, identities are valuable, and even malcontents have an incentive to protect them.

2.1.1 Growing Strength

We need the ability for identities to grow in strength as time passes. Computers tend to get faster at an exponential rate, so what is a socially acceptable key strength one year may be unacceptable the next. Fortunately, the salt string is independent from the key, so a user can generate new, stronger salt as he upgrades his computer, while retaining the same identity.

To find out how quickly a user needs to generate stronger salts, we need to investigate how hard it is for a principal (we use the term principal interchangably with the term user) to generate a salt of strength k. Besides this, we also consider the related question that if the principal has scanned n salts, how many of these salts should it communicate to its peer while transacting, so as to accurately represent the amount of work it has done. Let us approach these questions from a probabilistic view-point. All subsequent derivations that we do are based on expectations. The governing equation that determines the strength of a salt has a fixed number on the right hand side, which is the public key or the identity of the principal. This is compared to the output of a one-way hash function, the output of which is uniformly distributed over the entire range. Thus, the probability of a *salt* having at least strength i is $\frac{1}{2^i}$ (since the last i bits need to be the same). This answers the first question - to generate a salt of strength at least k, a minimum of 2^k salts need to be tried.

[1] Note that by equating a user with his public key, we sidestep many of the key management issues with public key systems, such as man-in-the-middle attacks. We are able to do so primarily because we use the keys for authentication, not secrecy.

The second question is concerned with finding the sweet-spot between sending more salts, and having the other peer know more accurately about the work a principal has done. If 2^k salts are tried, there is a definite chance of finding at least one salt with strength k. But, the number of salts with strength at least k is likely to remain the same even if $2^{(k+1)} - 1$ salts have been tried. Thus, if only the salt with the highest strength is sent, the estimate it provides is correct within a factor of 2 (This is because a single salt of strength k could have been obtained either when only 2^k salts had been tried, or when $2^{(k+1)} - 1$ salts had been tried). Thus, the probability of error in this case is $\frac{1}{2}$. Similarly, it can be shown that if a principal sends d of its best hashes (such that it does not happen that a salt of a particular strength is sent but another salt of the same strength is left out), the amount of work that its peer estimates can be at most off by a factor of $\frac{d}{d+1}$ from the real amount of work done. This is because sending a salt of strength k does not reveal anything about which salts have been tried (A more complete proof of the previous statement is mentioned in the appendix). Thus, to make the error probability additive instead of multiplicative, $O(n)$ salts need to be sent.

In practice, sending $O(n)$ hashes is not feasible because the overhead of transmitting the $O(n)$ hashes would be too high. Moreover, the receiving principal would be required to do an $O(n)$ amount of work to verify the message. We hope that principals would use a sliding window kind of thing, to send their best c salts, so that, the chances of error remain bounded by $\frac{c}{c+1}$.

2.1.2 Further Refinements

One complaint to make about the salt system described is that we assume that the salt was generated through the hard work of the identity's owner, when in fact there's no reason it couldn't have come from someone else. (Perhaps there is an evil mastermind with a supercomputer, who, out of the blackness of his heart, gives out strong salt to all of his favorite scoundrels.)

If we'd like for only the owner of the identity to be able to create salt, we can change our equation to the following:

$$\texttt{last}(\texttt{hash}(\mathit{salt}, \texttt{sign}_{prvkey}(\mathit{salt}), \mathit{pubkey}), b) = \texttt{last}(\mathit{pubkey}, b)$$

Now, we require that identifying token include a signed version of the salt. This cannot be generated without the private key, so a user would have to reveal his key in order for someone to help him generate salt.

However, this adds significant overhead to the size of an identity token and the ease with which we can check it. Since its benefit is dubious, we consider only the simplified earlier form in the remainder of this paper.

Now, having set up our notion of identity and the operations we support, we can begin to look at establishing and propagating trust.

3 Trust

The concept of what constitutes trust has been widely studied in areas like philosophy, economics, psychology and sociology. It has come to mean different things like personal trust, ("I trust that my parents would always do things for my own good"), trust because of good actions in the past, trust because of the perceived utility and trust because of one's perception about others. The notion of trust that we use in this work, is to associate a principal with the good/bad actions it has committed in the past, and to use past behavior as an indicator of the future behavior. Clearly, the idea of *identity* is central to this notion of trust, and depending on how identities in a system are modeled, we have the following trust architectures:

1. **Indistinguishable identities**: It requires but little reflection to see that trust cannot be modeled in a system with faceless principals, because principals cannot associate the good/bad behavior they have seen in the past with any other principal. An FTP server with username *anonymous* and password *guest* would be an example of this. A slightly improved version of this is one in which identities exist, but they are easy to create and fake. It is evident that this weak concept of identity is insufficient to model trust.

2. **Identity easy to create but difficult to fake**: In this system, it is possible to associate good actions with a principal, but there is nothing in this system to prevent a chiseler from committing misdeeds, creating a new identity and starting all over again with a clean slate. This is the familiar case of Slashdot *karma* wherein users are rewarded for their good deeds. The creation of

an identity is cheap, but building a reputation is difficult because reputation now depends on the good deeds that the principal has committed in the past. The major drawback of this kind of system is that it is unable to keep track of the misdeeds committed by a principal and hence, there is little incentive for good behavior unless the principal's rating is high.

3. **Identity difficult to create or fake**: This is a viable model for building and propagating trust as it can keep track of both the good as well as the bad actions committed by a principal in the past. Shysters can no longer run scot-free as every time they discard their old identity, they need to do a lot of work to create a new one.

4. **Identity certified by an oracle**: This is the familiar case of a certifying authority or an oracle, which is believed by everyone in the system, and it keeps track of behavior of all users in the system. If the trusted authority is infallible, then, clearly this is the most favorable case as far as building a trust model is concerned. But, in practice, having an infallible trusted authority is extremely difficult.

The present system that we propose falls in the third category. By making identity generation computationally expensive, we hope that even rogue principals would want to protect their identity. Though an infallible oracle would be nice to have, our solution is both practical and more powerful than categories 1 and 2.

Though it is possible to use our system to track both the good and the bad actions a principal has committed in the past, we record only the bad actions and tolerate nothing but strict compliance with the rules. Thus, the penalty for a misdeed is the death of the identity, irrespective of how much good work the principal had done earlier. We hope that such a high penalty for infraction would make principals less likely to commit any misdeeds. Our system puts the adage "Once a cheater, always a cheater" into practice.

For keeping track of the misdeeds of a principal, the principal needs to sign each message that it sends, using its identity. The signed message provides an irrefutable evidence of the mischief committed by the principal. Signing combined with our design requirement of the evidence being independently verifiable by any other principal means that trust, or rather, distrust is no longer subjective, or a matter of personal preferences, in our framework. It is a factual thing, since it is backed by concrete evidence of misdeed, which can be verified independently by any principal. Signing also takes care of the problem in which a trustworthy principal is unfairly implicated for some other principal's wrongdoings because a principal does not sign the content unless it can vouch for the authenticity of the content.

4 Example Framework

In this section we sketch an example framework which makes use of the ideas presented. We attempt to highlight the areas where our ideas help solve a problem in new ways, but don't concentrate on problems with well-known solutions.

Our example is based on the ConCert project[2], a peer-to-peer grid computing platform.

The ConCert network is made of a number of nodes spread across the internet. Any node on the network is allowed to offer tasks (machine code) for other nodes to do when they are idle. Certified code techniques such as proof-carrying code[9] are used to ensure that the code is safe to run, without the need for a certifying authority. This makes it impossible for miscreants to vandalize nodes by sending out malicious tasks. However, mischief makers can still cause trouble by offering to do a task, and then returning bogus results.[2]

There are a number of ways to deal with this. One is to give out only tasks for which the answer is easily verifiable. For instance, we can ask for factors of a number—we can easily check to see that the answer is correct. However, this restricts our range of tasks enormously. Another way is to send tasks to multiple parties and then compare the results. This wastes significant resources in the case that nobody is cheating. A third is to verify the results ourselves—but then we might as well have just done the work on our own in the first place.

In any case, after we detect cheating, we want to be able to do something about it. If we are using the system of identities described earlier, then we are able to blacklist this user, and they will not be able to trick us again without expending a lot of resources to generate a new identity. If we are using a trust propagation system based on evidence, then we can

[2]The situation is much worse if there is an economic reward for completing tasks!

alert the rest of the network of the user's misbehavior so that he'll have difficulty tricking anyone else, too.

Following is a sketch of how the system could use our ideas to make cheating much more difficult.

4.1 The Network

To begin, imagine a peer-to-peer network similar to Gnutella[5]. Nodes discover each other through broadcast ping messages and do "searches" for tasks to run. However, all messages are signed and accompanied by identifying tokens in order to enable our new features.

4.2 Nodes

Each node on the network has its own key that it generated before joining.

In addition, it maintains a blacklist of keys that it does not trust. With each key, it may optionally store evidence implicating the identity. These keys are stored in a hash table or other dictionary structure for fast lookup.

4.3 Detecting Cheating

We can use any of the methods above to detect cheating. Unless verification is essentially free (because correct answers are self-evident), nodes should do probabilistic verification, that is, check one out of every n answers at random. The probability of checking an answer depends on many factors, such as the importance of the job, the expected prevalence of cheaters, and the difficulty in checking results.

4.4 Evidence

The most obvious form of evidence is an incorrect result for a task. This consists of a piece of code and the purported result, signed by the identity in question.

Since verifying such evidence can be somewhat expensive (it requires re-running the task), one possible attack would be to flood the network with bogus "evidence" of cheating. Therefore, we want even an assertion that someone is cheating to be signed, as in, "I, Tom, certify that the program 0A4CB3… terminated in 91044 cycles and resulted in the answer 0380295…. However, Amit certified the following: …". Now, if someone receives this evidence and doesn't already know something about Tom or Amit, then by running the included code he can be assured of discovering that either Tom or Amit is lying. So, if Tom tries to flood the network with bogus evidence, then he will quickly be ignored himself.

Other kinds of evidence include task offers with broken certificates, and malformed packets.

4.5 Key Exchange

Establishing a connection consists of an exchange of nonces, as follows.

$$\begin{array}{cclll} \mathbf{A} & \rightarrow & \mathbf{B} & \text{salt}_A,\ \text{public}_A,\ \text{nonce}_A \\ \mathbf{B} & \rightarrow & \mathbf{A} & \text{salt}_B,\ \text{public}_B, \\ & & & \text{nonce}_B,\ \text{sign}_B(\text{nonce}_A) \\ \mathbf{A} & \rightarrow & \mathbf{B} & \text{sign}_A(\text{nonce}_B) \end{array}$$

The purpose of this handshake is to make it difficult for users without good identities to join the network. If **A** or **B** appears on the other's blacklist, or if their key strengths are inadequate, then the connection will be aborted. Since the nonces are generated randomly, an attacker **C** will not be able to simply replay old handshakes. Of course, it would be simple for **C** to secretly intermediate between **A** and **B** and eavesdrop, but this is acceptable because we are not attempting to provide secrecy. **C** cannot modify or forge any messages after the handshake because they will be signed by **A** and **B**; he will simply act as an invisible extra hop between them.

4.6 Requesting a Task

An idle node **A** can send out a request for work, which will be broadcast with a limited TTL. A node **B** receiving this message that has work to spare can respond by connecting to **A** and offering it a task. If **B** is in the idle node's blacklist, it should certainly refuse. Furthermore, the fact that the idle node received a response from **B** means that they are close in the network and that **B** is connected to some node that doesn't know he is a cheater.[3] Therefore, it makes sense for **A** to broadcast his evidence of **B**'s misbehavior to his local neighborhood.

[3] Or connected to a node that is passively assisting it by ignoring evidence. See the section on future work for details.

4.7 Propagating Evidence

In fact, there are several circumstances in which we send out evidence. When we first discover that a node has reported a wrong answer, or provided false evidence (or committed some other infraction), we assume that it has just "gone maverick" and we want to alert others as soon as possible to minimize the damage it can do. Therefore, we flood the network with our new evidence. In order to avoid a huge amount of flooding of evidence in the case that a rogue node tries to trick many other nodes simultaneously, a node should not forward an evidence flood if it has received evidence about the same identity within some short period of time.

In a real peer-to-peer system, the network topology is always changing, with new users being added and old users leaving. Therefore, telling the network about fraud once will not be enough.

If we notice a message pass through us that came from a blacklisted node or includes a blacklisted node in its source route, then as above we assume the node must be nearby and we send out a TTL-limited broadcast with evidence implicating that node.

Finally, when we first join the network, we should broadcast some random subset of our blacklist evidence. This is to prevent evidence from being "forgotten" by the network because there was never any occasion to share it (the identity never attempted to connect to the network). The amount that we broadcast would depend on this attrition rate and how much we desire to retain possibly obsolete evidence.

Though this appears to be a lot of broadcasting, remember that the amount of evidence around is limited by the number of identities generated, and we have made identity generation difficult. To generate a single flood of the network, a vandal must spend several CPU-hours to make an acceptable key. In this light, the flooding of evidence seems insignificant even compared to the broadcast PING messages common in peer-to-peer protocols!

5 Other Applications

In general, our idea of identity generation is applicable whenever there is a desire for users on a network to identify other users, and a guarantee of uniqueness is not necessary. (For instance, it would not be appropriate for an electronic voting system.) Our notion of trust can be applied whenever there is an appropriate system of "evidence", for instance in a network where there are queries and results that can be objectively verified. Many existing peer-to-peer systems fit both of these criteria:

5.1 Caching Hierarchies

The benefits of a cache hierarchy have been well studied[3]. Having a caching hierarchy essentially increases the effective user population and therefore, a caching hierarchy has a higher hit rate than a single stand-alone cache. When the caches in the hierarchy are close enough, this can result in substantial reduction in end-user latency and sizeable bandwidth savings to the other parts of the Internet. Moreover, all requests that are satisfied by the caches, do not need to go to the origin server, and thus, the load on the origin server is reduced. In spite of these benefits, one of the main reasons for such hierarchies not being common is the lack of trust across autonomous systems. Our framework is able to address this problem (though it might be overkill!).

A more interesting case arises when we consider caching by end-users as a way to deal with flash crowds, akin to CoopNet[16]. There is a major difference from CoopNet though. End-users in our system would query their peers first, as opposed to CoopNet, where clients usually first go to the origin server and are then redirected to one of the client machines. Accessing the origin server in the scenario we envision would be necessary only to verify the authenticity of the content.

Most of the steps to be followed in this case are the same as detailed with regard to the ConCert project. Thus, to participate, each cache generates a key pair that represents its identity. Each cache also maintains a list of blacklisted identities. To verify the authenticity of a response, a cache can either compare the different results for its query against each other or the cache can simply fetch the required document from the origin server and compare it with the response it has obtained from its peers. [4]

In a nutshell, we propose enhancing ICP or HTCP to do a key exchange (handshake) and then exchange signed content. Caches that interact frequently with each other can maintain a persistent

[4] Since many web pages are not static, this may require an extension to HTTP that allows a cache server to fetch a page (or at least its hash) at a specific time in the past in order to verify old evidence of lying.

connection, so that the connection overhead cost is amortized.

5.2 Freenet

Our scheme is directly applicable to Freenet[15], and highly compatible in spirit (we preserve anonymity and use strong cryptography).

In Freenet, content hash keys (CHKs) are a way of requesting files based on a hash of their content. When receiving a file, we have a simple way of verifying that it it's correct: compute the hash of the file and compare. Other keying schemes also have potential for verification, such as keyword signed keys (KSKs). Wherever verification is possible, we have the potential to blacklist anyone who tries to pass off incorrect versions of files. (Unfortunately, evidence can be extremely large, since we must include the entire signed file that they sent!)

There are of course many other issues to consider. However, we believe that our ideas could be incorporated in such a way as to make Freenet even more robust against attack.

5.3 File-sharing systems

Current file-sharing systems have a very weak concept of identity. The identity of a user is her username, and nothing stops a user from creating new identities. Clearly, there is very little trust that is possible. Our scheme can be applied to this system so that identities no longer remain as fluid, and it is easy to associate good/bad actions with a user.

The only big hindrance in our scheme being applicable to this framework is that there is no clear mechanism for verifying the content. For example, whether a MP3 song has a good enough quality or not is subjective. This is indeed a tough problem, but could be solved if there were public databases of songs stored along with their hashes on the Internet. This idea is similar to that mentioned in [17]—the notion of caching trust rather than content.

Besides this, there is a small hindrance. There might be users in a peer-to-peer system who might just want to download stuff from the network and would not want to bother spending their resources in computing the required identity. This means that such users even if they have some content cannot act as senders for that content. One way to deal with such a problem is to require both parties in a transaction, instead of just the sender, to have sufficiently strong identities. This way all users in the network could be made accountable for their deeds. Another rationale for doing this is that *good* users would like to help only the good users, and not a user with a questionable track-record.

6 Related Work

Most existing literature in the area of trust inherently assume the existence of a certifying authority, and their main focus is to develop formal models for building and propagating trust, that are computationally tractable. Since this itself is a hard problem, researchers have not looked on to the next logical step i.e., modeling trust without the assumption of an oracle to vouch for the conduct of all principals.

Marsh [7], was probably the first to formalize the concept of trust as a computational concept. His model tried to incorporate the varied aspects of trust, as detailed in the huge amounts of literature on trust in economics, philosophy, psychology and sociology. The resulting model was therefore fairly complex as it included many aspects of social trust, and depended on many variables that represented abstract notions such as 'risk', 'competence', 'possible gains by cooperation' and the 'importance of the task to the agent'. In his other work[8], Marsh also compares the optimistic, pessimistic and realistic model of trusts and concludes that a model of trust based on realism is better than being uniformly and naively optimistic or pessimistic about everyone else.

Abdul-Rahman[12] et al. discuss the idea of supporting trust in virtual communities and develop a trust model, in which trust propagates in a fashion akin to the real world, i.e., through word-of-mouth. They have *recommenders* in the system, and an agent may decide how much it trusts a recommender and the suggestions given by the recommender about an agent.

The focus of the above works is completely orthogonal to what we propose in this paper. They focus on how to model and propagate trust, in the presence of an all powerful central authority, where as we present an alternative to having a central authority, and show how not having a central authority does not diminish in any way, our ability to model trust. Thus, most of their models for trust and trust propagation are easily applicable to our framework. There is one difference though. They have even considered subjective trust, which is not true in our case because of the requirement that all messages be signed. Thus, trust propagation in our scenario becomes very easy, because a principal can present concrete evidence of an identity's unworthiness.

Hopper et al. [10] detail a service discovery and delivery system, in which clients can receive guaranteed delivery of multimedia content from a content provider, in return for payment. Clients in their setting do not actually care about which content provider is providing the service. To ensure that neither the client nor the content provider can cheat, they have a concept of *verifier* which is trusted by all parties of the system.

Meyers et al. [11] talk about security in a system, in which a publisher-centric cache executes code on behalf of an origin server. Their system is based on a certifying authority, using which all the publishers and caches are hierarchically certified. Publishers can ship out code, that can then be executed at one of the caches authorized by the publisher. In addition, to check whether the cache is behaving correctly or not, a client can probabilistically verify the output it received from a cache with the origin server.

Both of the above works end up using the concept of a central authority, and so, they do not need to have identities that are computationally difficult to generate. We can model both the above works in our framework, and make them to work without using the concept of a central authority. Apart from this, both the projects mentioned above use signing of messages to catch miscreants and use the idea of probabilistic verification to reduce the system overhead. We use both these techniques in our system for essentially the same purposes. Thus, our idea of having a computationally intensive identity, that evolves with time, is what separates our work from both the above mentioned works.

Reference [6] sketches a system, called Hash Cash, for "charging" for email delivery by requiring senders to compute partial collisions for hash functions. This idea of auditable metering (a checkable certificate of some amount of work done) was also seen in [13] and [14].

These papers all use auditable metering for one-time use, essentially slowing down a system from generating requests to a service extremely quickly. Our system is the first to generate a single token that is re-used, and to equate this token with the idea of an identity.

7 Future Work

Certain kinds of malicious behavior are not easy to provide evidence for, even if they are easy to detect. It would be interesting to design a system where more types of misbehavior could be captured.

For instance, assume we were able to synchronize a clock across the entire network. Then, suppose we require that all broadcast packets contain a timestamp, and that any packet with a timestamp outside the normal error range for the clock be dropped. Now, we can provide evidence for a user who rapidly floods the network with broadcast traffic: a collection of packets with close-together timestamps.

Another way that nodes can misbehave in our current system without being penalized is to passively assist blacklisted nodes. For instance, suppose that **C** is on **A**'s blacklist, but is connected to the network through **B**. **A** can present **B** with evidence of **C**'s wrongdoing, but **B** has no obligation to pay any attention to it; he can continue to forward on **C**'s messages. Though **A** can place **B** in his personal blacklist, he has no way of formulating irrefutable evidence that **B** is assisting **C**. It's hard to imagine a system with un-ignorable evidence, but it might be possible.

There is significant overhead associated with signing each message and including a key and salt along with it. Though the overhead is "just" a constant factor, investigating ways of reducing it (perhaps by caching the keys and transmitting only hashes of them) would probably make our scheme more practical.

8 Conclusion

We have proposed a new system for generating identities whose strength can be easily independently verified. Armed with these identities, we have proposed a simple system of trust based on irrefutable evidence, and given an example of how such a system could be used to solve the problem of forged results in a peer-to-peer grid computing application. In doing so, we retain the essential peer-to-peer properties of anonymity and decentralization. We have also suggested several other applications where our ideas may prove useful.

9 Bibliography

References

[1] KaZaA and Morpheus, http://www.fasttrack.nu/

[2] Harper et al., http://www.cs.cmu.edu/˜concert/papers/proposal/proposal.pdf

[3] Wolman et al., On the scale and performance of cooperative Web Proxy Caching

[4] Schneier, Applied Cryptography, second edition.

[5] http://www.gnutella.com/

[6] http://www.cypherspace.org/˜adam/hashcash/

[7] Marsh, Formalizing Trust as a Computational Concept, Ph.D. thesis, Department of Computing Science and Mathematics, University of Stirling

[8] Marsh, Optimism and Pessimism in Trust, Technical Report CSM-117.

[9] Necula, Lee et al., Safe, Untrusted Agents using Proof-Carrying Code.

[10] Hopper et al., Trusted Service Discovery and Delivery, 15-744 Course Project.

[11] Myers et al., A Secure Publisher-Centric Web Caching Infrastructure, published in IEEE Infocom '01.

[12] Abdul-Rahman et al., Supporting Trust in Virtual Communities, In Proceedings Hawaii International Conference on System Sciences 33, Maui, Hawaii, 4-7 January 2000.

[13] http://www.wisdom.weizmann.ac.il/˜naor/ onpub.html - Pricing via Processing - Proceedings of Crypto 92 - Cynthia Dwork and Moni Naor

[14] http://www.cs.huji.ac.il/˜dalia/pubs.html - Auditable Metering with Lightweight Security, Financial Cryptography 97, Dahlia Malkhi and Matt Franklin

[15] Clarke et al., Freenet: A Distributed Anonymous Information Storage and Retrieval System.

[16] Padmanabhan et al., The Case for Cooperative Networking, published in IPTPS '02.

[17] Satyanarayanan, Caching Trust Rather Than Content, Operating Systems Review, Volume 34, No. 4, October 2000.

10 Appendix: Analysis of Salt

Let us first describe the problem more formally. Suppose a principal P has tried n salts, and in its interaction with Q, suppose it decides to send its best k salts (such that it does not happen that a salt of a particular strength is sent but another salt of the same strength is left out). Based on the strength of the k salts that it receives, Q expects that P would have carried out m hashes. The question we want to ask is how does the relation between m and n vary with increasing k, in the worst case? By worst case, we mean that for a given k, what is the largest n possible?

The derivation here is based on the idea of expectations. Let $P(i)$ denote the probability that a salt is of strength at least i, let $S(i)$ denote their collection and let $PN(i)$ be the corresponding number of salts. Let $QN(i)$ denote the number of salts with strength exactly i. Thus, $QN(i)$=$PN(i)$ - $PN(i+1)$. $P(i)$ is equal to $\frac{1}{2^i}$. Clearly, $S(i+1) \subset S(i)$. We are interested in finding $PN(0)$ based on the information about the higher order nodes in our system.

Let each salt that is tried in our system be represented as a $S(0)$ node (since all salts have strength at least 0). The tree is then built recursively - if there are two $S(i)$ nodes that are free (by free we mean that they have no parent), they join together and cease to be free to form a $S(i+1)$ node, unless for each i at most one $S(i)$ can be free. The rationale behind having two $S(i)$ join together to form one $S(i+1)$ is that $P(i+1)$ is exactly equal to one-half $P(i)$. Intuitively, for every two distinct $S(i)$ nodes that we find, one of them is expected to belong to $S(i+1)$.

Let c be the highest number for which $S(c) \neq \phi$. Therefore, $PN(c) = 1$. If we just send the sole element of $S(c)$, it conveys the fact that the principal has tried at least 2^c salts, whereas the actual number of salts tried could be anywhere from 2^c to $2^{(c+1)} - 1$, thus, implying an error probability of $\frac{1}{2}$. It is also worth noting that sending elements of $S(c)$ gives us no indication as to what the elements of $S(c-1)$ are. If we decide to send lower elements from the set $S(c-i)$, then we need to send at least 2^i elements from set $S(c-i)$, to just cover the work represented by one element of $S(c)$. Thus, if we send all the elements of $S(c-i)$, $m = NP(c-i) * 2^{(c-i)}$, where as the actual number of salts could be anywhere between $NP(c-i) * 2^{(c-i)}$ to $NP(c-i) * 2^{(c-i)} + 2^{(c-i)} - 1$. Since $NP(c-i)$ would be at least 2^i, thus the error probability is at most $\frac{2^i}{2^i+1}$. Thus, $\frac{m}{n} \leq \frac{k}{k+1}$, if we send just all nodes at a level. $\surd$