Name Privacy and Malleability

June 28, 2016

1 Introduction

ICN architectures such as CCN and NDN were designed with the following principle: avoid resolution services such as DNS to learn routable information. (At least for NDN, this principle has since been violated with the development of the NDNS - the name data name service.) This means that any consumer in possession of the right data name can request it from the network without first querying some resolution service. If nothing is done, application data names are conveyed and propagated in the network. While this allows requests to be routed, it does so at the significant cost of privacy. Specifically, the name of a request and its response reveals exactly what is sought after. Privacy, in this context, means that a request name should have no correlation with the data carried in the response. But how can we achieve this type of privacy without a resolution service? And if it can be done, can it be done *efficiently* and, ideally, without placing trust in routers? In this note we seek to answer these questions.

2 Main Goal

Since there is no resolution service and we cannot simply mimic application names in network packets (since, by our definition, that is not private), any private naming scheme must involve some translation function that locally maps application data names to network names. We denote this function as $T(N)$ for some name N. Our expectation is that a request with the name $T(N)$ leaks nothing about the data carried in the response. Moreover, we want $T(N)$ to reveal nothing about N. Specifically, the desired security goal is that for a given name N, the probability for any probabilistic polynomial time adversary to distinguish the transformed version of N – $T(N)$ – from a random string is negligible (in something). This implies that the distribution $(T(N), T(N))$ for a fixed N is computationally indistinguishable from the tuple $(T(N), r)$ for the same N and random r. We therefore assume that $T(N)$ is a probabilistic algorithm, otherwise distinguishing would be trivial.

Put together, we want $T(N)$ to have the following properties:

- An adversary cannot deduce N from $T(N)$.
- An adversary cannot deduce N from the data carried in the response of $T(N)$.

- The intended producer of N can recover N from $T(N)$.

This does not work in the group Zp

- $\mathsf{Init}(1^\lambda)$: Generate and output a finite cyclic group G with generator g.
- $\mathsf{KeyGen}(G, g)$: Generate a random $k \in G$, compute $p = g^k$, and return (k, p).
- $\mathsf{Hash}(G, g, m, p)$: Generate a random $r \in G$, compute $\alpha = p^r$ and $\beta = g^{m+r}$, and return (α, β).
- $\mathsf{Compare}(G, k, (\alpha_1, \beta_1), (\alpha_2, \beta_2))$: Compute $v_1 = \beta_1^k \alpha_2$ and $v_2 = \beta_2^k \alpha_1$. Return 1 if $v_1 = v_2$ and 0 otherwise.

This discrete-log anonymous hashing scheme (DL-AHS) is deemed correct if for all $(G, g) \leftarrow \mathsf{Init}(1^\lambda)$, $(k, p) \leftarrow \mathsf{KeyGen}(G, g)$, and all $m_1, m_2 \in G$ it holds that

$$\mathsf{Compare}(G, k, \mathsf{Hash}(G, g, m_1, p), \mathsf{Hash}(G, g, m_1, p)) = 0$$

and

$$\mathsf{Compare}(G, k, \mathsf{Hash}(G, g, m_1, p), \mathsf{Hash}(G, g, m_2, p)) = 1$$

hold with negligible probability.

Definition 1. *An AHS is secure if it is computationally intractible for any probabilistic polynomial-time adversary to (a) recover a message m from $\mathsf{Hash}(G, g, m, p)$ given $(G, g) \leftarrow \mathsf{Init}(1^\lambda)$ and p from $(k, p) \leftarrow \mathsf{KeyGen}(G, g)$ or distinguish $\mathsf{Hash}(G, g, m, p)$ from the tuple (g^{r_1}, g^{r_2}) for random $r_1, r_2 \in G$.*

Theorem 1. *The DL-AHS scheme is secure.*

Proof. The first part of the proof follows from the hardness of the Discrete Logarithm (DL) problem. The output of $\mathsf{Hash}(G, g, m, p)$ is a tuple $(\alpha, \beta) = (p^r, g^{m+r})$. It follows that learning r from either α or β is intractable.

The second part of the proof requires us to show that (α, β) is computaionally indistinguishable from the tuple (g^{r_1}, g^{r_2}) for random $r_1, r_2 \in G$. To do this, assume there existed such a distinguisher D that could distinguish the target tuple from a random one with non-negligible probability. By the properties of r used to generate (α, β), D could then be used to solve the DDH with the same probability. To do so, D would work as follows.

show the reduction

□

Assume that a node had some data structure with two procedures: insert and lookup. These procedures work as follows:

- insert: Given a name N and value v, insert an element into the data structure which maps to v if not already present.

- lookup: Given a name N, return the value v associated with it in the data structure.

We do not specify how they are implemented. Let k be the number of unique elements in this data structure at any given point in time. We will prove that their respective runtimes must be $O(1)$ and $O(k)$, respectively.

Theorem 2. *Let D be a data structure as defined above. Its* insert *operation runs in* $\Omega(1)$ *time.*

Proof. TODO: hash table. constant time insertion. □

Theorem 3. *Let D be a data structure as defined above. Its* lookup *operation runs in* $\Theta(k)$ *time.*

Proof. TODO: every element is indistinguishable from the rest. comparison by exact match will not work, must run the comparison over every element in the database. done.

proof by reduction: - assume that some implementation did exist that ran in time less than k. - show how this could be used to distinguish between the two distributions above. - for the valid one, the time would be less than k - for the invalid one, the time must be k (since all must be checked) □

3 Onions vs Sponges

Sponges allow a node to route a packet without having any pre-allocated state with the node. Also, onions ultimately reveal the plaintext to some entry in the circuit (the exit node). Conversely, sponges reveal nothing about the name to each node. However, onions can be forwarded in (roughly) constant time.

Question: are there cases where sponges would be forwarded just as fast as onions? (with respect to total end-to-end forwarding delay...)

Onion: requires h computations for a circuit of length h – average length of circuit is at most 5 hops (?) (this is small!) Sponge: depends on the namespace tree... the size of the tree cut (include each node in that tree)

Question: so if sponges are less efficient and induce more network overhead, why would we use them? Answer: you don't! Encapsulation is probably the *best* way to go from an efficiency perspective Implication: Supporting both caching and private requests (efficiently) is not possible

4 Relation to PIR

TODO: we examine the CPIR model since IPIR assumes multiple non-colluding servers. Here, we assume a sequence of individual servers (caches). Moreover, we want the protocol to run in a single round.