Naming in Content-Oriented Architectures

Ali Ghodsi
UC Berkeley
alig@eecs.berkeley.edu

Teemu Koponen
Nicira Networks
koponen@nicira.com

Jarno Rajahalme
Nokia Siemens Networks
jarno.rajahalme@nsn.com

Pasi Sarolahti
Aalto University
pasi.sarolahti@iki.fi

Scott Shenker
UC Berkeley & ICSI
shenker@icsi.berkeley.edu

ABSTRACT

There have been several recent proposals for content-oriented network architectures whose underlying mechanisms are surprisingly similar in spirit, but which differ in many details. In this paper we step back from the mechanistic details and focus only on the area where the these approaches have a fundamental difference: naming. In particular, some designs adopt a hierarchical, human-readable names, whereas others use self-certifying names. When discussing a network architecture, three of the most important requirements are security, scalability, and flexibility. In this paper we examine the two different naming approaches in terms of these three basic goals.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design

General Terms

Design, Documentation, Security

Keywords

content, naming, scalability, and security.

1. INTRODUCTION

There have been several recent proposals for content-oriented network architectures[1] whose underlying protocols are surprisingly similar in spirit, but which differ in many details. In this paper we step back from the minor mechanistic differences and focus only on the one area where these proposals have a fundamental difference: naming. In particular, some designs adopt hierarchical, human-readable names, whereas others use flat, self-certifying names. There is an ongoing debate in the community (see [10]) about which approach is most appropriate, but this debate has produced more heat than light. To help clarify the issues around naming, and thereby move this debate forward, in this paper we examine the two different naming approaches in terms of three basic architectural goals: security, scalability, and flexibility.

Content-oriented architectures differ from today's architecture in several fundamental ways, and it is useful to review the relevant differences before plunging into the main content of this paper. Data retrieval in today's architecture is typically done through HTTP, which uses URLs as names for content; to fetch data a client sends a request to the host associated with the URL. The requester is assured the data is valid by using protocols that ensure that they are talking to the host named in the URL. Scalability follows from the hierarchical structure of DNS names (and, at the IP layer, the aggregation of prefixes). The design is flexible in that the data path makes no assumptions about the naming model.

In content-oriented architectures, content is named directly, placed in the network by "publishers", replicated in caches by the network, and requested by name through a find or fetch primitive. Finding a copy of the desired data object is accomplished through the equivalent of a name-based anycast distribution of the request, and the object can be returned from any host (or network element) holding a copy. The validity of the object must therefore be ascertained directly, not by verifying which host supplied the object. This need for direct object verification was the original motivation for using self-certifying names in content-oriented architectures.[2] By "self-certifying name" we mean a name that (together with some signed metadata) allows direct verification of the binding between the name and an associated object, as will be described in more detail in Section 2.

However, the use of self-certifying names has been recently criticized on two grounds in particular (see [10] for a critique of this approach and [12] for a response). These objections can be roughly summarized as follows:

- Scalability: because these names are not hierarchical, they cannot handle the necessary scales.
- Security: self-certifying names require another service to translate between human-friendly and self-certifying names, so any security benefits are nullified by weaknesses in such a translation service.

In this paper we respond to these objections. We argue that, in fact, self-certifying names (i) have better security properties than human-readable names because they can deal with denial of service without greatly limiting the nature of trust mechanisms and (ii) can have better scalability properties than hierarchical names by allowing more flexible aggregation. Thus, based on the

[1] See [1, 2, 4–6, 8, 13] for a small sampling of the recent literature, which is too large to provide a comprehensive overview of here.

[2] See [14] for an early discussion of self-certification in "named data".

SIGCOMM ICN'11, August 19, 2011, Toronto, Ontario, Canada.

architectural goals of security, scalability, and flexibility, we believe that self-certifying names are the correct choice. However, we also observe that the two naming schemes are not as different as commonly believed, and can be combined quite easily.

This paper is structured as follows. We discuss the relationship between naming and security in Section 2, and analyze scalability in Section 3. We review approaches to trust management in Section 4 and we conclude in Section 5.

2. NAMING AND SECURITY

Network security is often broken down into four components: integrity, confidentiality, provenance, and availability. That is, the security measures in the network infrastructure and end-host network stacks must ensure that data is delivered reliably (availability), comes from the appropriate original source (provenance), and has not been tampered with (integrity) or read by others (confidentiality).

The latter three of these can be handled on an end-to-end basis using basic cryptography, once cryptographic keys are properly bound to the appropriate entities (as discussed below). In contrast, availability is the responsibility of the network itself and must be addressed on two different levels. First, the underlying network infrastructure must reliably deliver bits, which requires that its components cannot be easily compromised and data delivery cannot be overwhelmed by denial-of-service attacks; this is completely independent of the naming system, so we do not consider it further here. Second, the infrastructure must provide protection against content-level denial-of-service attacks; *i.e.*, even if bits are delivered reliably, the network must ensure that the delivered object has the correct name and provenance, or else requesters could be denied the object they desire. Naming does play a critical role here, and our subsequent discussion of availability will focus only on this aspect of denial of service.

2.1 The Basic Bindings

As in the SDSI/SPKI framework [3, 9], in content-oriented networks each piece of mutable data is associated with a "principal" who is responsible for its content (either because the principal created it, verified it, or merely because the principal vouches for it). When we talk about the provenance of an object, we are referring to this principal.[3] To effectively use cryptography to achieve our security goals, for each object we must establish bindings between three different entities:

- *Real-World Identity (RWI)*: This is the real-world entity who is the principal for the data. For example, a real-world identity may refer to a person or organization.
- *Name*: This is the name handed to the network in the fetch-by-name primitive, and must be sufficient to identify the object within the network. Principals are responsible for naming their content.
- *Public Key*: Each principal is associated with a public-private key pair. This is the cryptographic key that can be used by the receiver to verify that the RWI did indeed sign the content.[4]

[3]Immutable data can be checked against a cryptographic hash of the content, and is therefore easier to deal with, so we ignore it in the following discussion.

[4]We do not consider the particular cryptographic algorithms used, how agreement on them is reached, or how they can be changed over time; suffice it to say here that these are not insurmountable barriers, but do require careful thought.

All the three potential bindings between the above entities (RWI–name, RWI–public key, and public key–name) have an essential role in verifying the provenance of data.[5] If the public key is not bound (directly or indirectly, as we discuss below) with the corresponding RWI, then anyone could claim to be the principal associated with the data; that is, principals with keys k_1 and k_2 could both claim to be CNN, and without a key–RWI binding a user cannot know which claim to believe. If the name is not bound to the RWI, then it is impossible to identify the principal of the object; if someone claims that fetching foobar will return CNN headlines, but there is no cryptographic connection between foobar and CNN, then there is no way to validate that assertion. Finally, if the name is not bound to the public key, then the receiver doesn't know which public key to use to verify the provenance of the data: that is, if a user fetches the object named foobar, but the name foobar is not bound to any cryptographic key, then the user has no way of cryptographically verifying anything about the returned object (in particular, the user cannot verify that the object is indeed named foobar).

Because bindings are transitive, one does not need to directly ensure all three bindings: providing any two of them implies the third. The question, then, is how are these two basic bindings established, and it turns out that the two naming schemes we have discussed establish the bindings in rather different ways.

2.2 Binding in Naming Schemes

With human-readable, hierarchical naming (as used in *e.g.*, [5]), the entire name is specified as free-form hierarchy and the name (much like DNS names) is a sequence of human-readable strings. The purpose of human-readability is to establish an intrinsic binding between the name and the RWI. The second binding (between the name and key) can be established using an external authority such as a Certificate Authority (CA). In other words, in our example above, the string "CNN" in the name would be sufficient for the user to determine that the RWI associated with the name is CNN, and a CA will verify that a key belongs to the name containing CNN.

For convenience, we will assume that self-certifying names for *mutable* data consist of a cryptographic hash of a public key (principal identifier) and a globally unique label determined by the RWI, and both are of fixed size.[6] That is, names have the form **P:L**, where P is a cryptographic hash of the principal's public key and L is the label. The use of cryptographic hash function for computing P provides the binding between the name and the key, by enabling the receiver to check that the key indeed hashed to P. That is, if someone claims that a key is associated with a name P, then simply taking the hash confirms it; the existence of a binding does not mean that knowing P is enough to derive the key, it is merely enough to verify a key.[7] However, the use of such hash functions renders the name unreadable to humans, so the binding between the RWI and key must be established by an external authority.

Note that in both naming approaches, one binding is intrinsic to the naming system, while the other must be supplied by an

[5]The other two properties, integrity and confidentiality, do not require all three, so we do not discuss these properties further.

[6]There are other ways to construct self-certifying names, but they are logically equivalent. Also, note again how the naming of *immutable* data may simply rely on cryptographic content hashes.

[7]Each object consists of the following: <data, public key, L, metadata, signature>. This is enough information to verify the name-object binding, so the receiver can know that a delivered object does indeed have the appropriate name. By omitting the label L inside the signature, the text in [6] is in error on this point.

outside authority. See Figure 1 for a depiction of the bindings. Because these externally provided bindings are similar in spirit (whether they be for RWI–key, for self-certifying names, or name–key for human-readable names), the need for, and the nature of, these external mechanisms is not a deciding factor between these two naming systems. We will call systems that bind keys to RWIs or names *trust mechanisms*. These mechanisms don't mean you actually trust the entity or person to be honest, only that you trust that they are who they say they are.

There are obvious and well-known drawbacks with both naming schemes. Self-certifying names pose a challenge to usability, since humans cannot understand or remember them.[8] In addition, any move to such names would require significant reworking of the architecture (which, because this is a clean-slate discussion, is not seen as disqualifying them, but it is a serious drawback).[9] Finally, cryptographic algorithm upgrades will result in name changes, and careful engineering is required to manage their usability implications.

In contrast, human-readable, hierarchical names are more usable, more backwards-compatible, and remain unchanged while cryptographic algorithms evolve, but they provide a much weaker intrinsic binding. While the intrinsic name-key binding is cryptographically tight in self-certifying names, the intrinsic binding between name and RWI in human-readable names is far from perfect because it relies on the name being well-understood and unambiguous. This clearly does not always hold; as an example of a potentially ambiguous name, "ICSI" simultaneously refers to the Institute for Clinical Systems Improvement, the Institute of Company Secretaries in India, and the International Computer Science Institute. There is no way for a reader encountering this acronym in an object name to make this distinction without some external guidance. Even worse, phishing attacks try to confuse users with names similar to real ones; users frequently make mistakes in separating misspelled or names in different character sets from real names. Moreover, tiny URLs completely obliterate the name–RWI binding. In addition, human-readability creates contention in the namespace; when names are meaningful to users, some names become more desirable than the others.

Thus, both systems have serious disadvantages, but none of these flaws are fatal. However, the two naming systems have very different ways of dealing with denial of service, and this represents a more fundamental difference between the two.

Denial of Service

If the network cannot determine which objects are legitimately associated with a given name, then attackers can launch a denial of service attack by having many hosts claim to have data associated with that name. Thus, to protect the receiver from receiving[10] (possibly large amounts of) false content, the network needs to know the binding between the name and the key so it can verify that a given object is associated with a given name. However, the network does not need to know the RWI; all the network needs to do is reliably map names into objects associated with that name, while the receiver is left with a responsibility to use a name that corresponds to the RWI it finds trustworthy (and not trust to content referals from questionable sources). We'll return to the trustworthiness in Section 4.

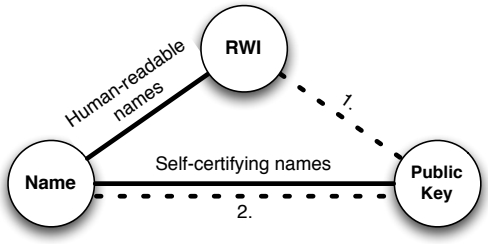

Figure 1: A depiction of the three entities and the different bindings between them. Two naming schemes provide different intrinsic bindings (solid lines) but require both an external authority to provide one additional binding (dashed line): with self-certifying names it's the binding (1), whereas with human-readable names it's the binding (2).

In self-certifying names, the binding between the name and key is intrinsic and strong: given a key, one can reliably (within the certainty of modern cryptography) determine if a name is associated with that key. On the other hand, in human-readable names, the binding between the key and name is not inherent in the name itself, so the network must obtain that binding somehow. How this occurs depends on when in the retrieval process the validation occurs; there are two possibilities.

First, the data can be validated when it is first published.[11] With self-certifying names, the network has the necessary verification tools to do this; it can require any publisher of data to provide the necessary signatures that associate the object with the name, as the key and name are intrinsically bound. For human-readable names, the network needs to use an external mechanism to bind the name to a key (which can then be used to check the signatures provided by publishers). For this, the network needs to know about and understand the external trust mechanisms relevant for that particular principal. This greatly limits the nature of the external trust mechanisms that can be employed. In practice, we argue this translates to a requirement of deploying a global PKI, which has its known issues that range between practical management challenges to almost Orwellian concerns [11].

Second, the data can be validated at fetch time, with the key provided within the request itself; that is, the binding between name and key is provided by the requester (who presumably could know about and understand the relevant trust mechanisms). In this case, the requester supplies the key (or its hash) along with the name in her request (and can sign them, so that this binding can't be tampered with). For this to work, content-oriented routing must be done on a name-key basis, not just on the name, in order for routing requests to reach objects associated with the appropriate key. This makes the key an essential part of the name (because it is used for routing); the name is no longer a pure human-readable string but includes a cryptographic part. The name has now become self-certifying, taking the form of P:L, with P being (as before) the hash of the public key and the label L being the hierarchical, human-readable name. The debate, then, is not about

[8] However, see [14] for a discussion of usability.

[9] The DOI project is an interesting experiment in incremental deployment of a flat naming structure.

[10] Here we assume that the receiver can tell that the delivered object is false, but our concern is that the network never delivers the correct data because it has responded to the request with a false object; this can be used as a denial of service attack.

[11] A host can "publish" content by telling the network that it is willing to provide copies of an object with a certain name. The question we are discussing here is what kinds of cryptographic checks are used to verify that the object is indeed associated with that name.

Figure 2: In deepest match, an exact match for each part of the name is looked for. The matching begins from the end of the name (above D) and proceeds a part by part to the beginning (A), until a match or there's nothing to look for.

human readable versus self-certifying, it is about what kinds of self-certifying names should be used. This hybrid choice merely requires that the labels in the P:L format for self-certifying names be human readable and hierarchical.

It is worthwhile to note that delaying the validation of the data until after the data is delivered is not an option: validation *after* fetching results in denial of service, as the requester may never receive a valid copy.

This completes our discussion of the security concerns, which we feel drives us towards self-certifying names. We now must confront two remaining issues: scalability and flexibility. We do so in the next two sections.

3. NAMING AND SCALABILITY

In this section, we examine the scalability of naming.

3.1 Explicit Aggregation

Hierarchical names help scalability by reducing the size and update-rate of the routing tables. While this is well-known, it is instructive to walk through the semantics of hierarchical routing in more detail. Consider a name of the form com.CNN.headlines. In terms of routing semantics, this name means that if you follow routing entries to com, you are guaranteed to find an entry for com.CNN somewhere along your path, and similarly if you follow routing entries to com.CNN you will eventually find an entry for com.CNN.headlines. These semantics, not any other details about hierarchies, are what enable scaling.

In terms of global uniqueness, each entry in the hierarchy is not guaranteed to be unique, but each prefix is (*i.e.*, com, com.CNN, and com.CNN.headlines are all globally unique). Because of this, one cannot route to arbitrary fragments (*i.e.*, headlines, or CNN.headlines) but must look for prefixes to be assured you are routing towards the right entity. It is the lack of global uniqueness of fragments, not the need for aggregation, that drives the use of longest-prefix-match.

The common assumption is that aggregation is impossible with flat naming.[12] It is true that names do not build in hierarchy, so aggregation does not simply fall out of the naming structure itself, but it turns out that this lack of inherent hierarchy provides greater flexibility in aggregation.

Given a set of globally unique names (say, A, B, C, etc., each of the form P:L), one can construct "explicit aggregation" by using concatenations of the form A.B.C.[13] The semantics of such concatenations are that when following routing entries for A, you will eventually find one for B; and that when following routing entries for B, you will eventually find an entry for C. We will call this the *aggregation invariant*. There are two questions to address:

How do you route using concatenations? The routing table consists of individual names A, B, etc., and when confronted with concatenated name A.B.C.D, the router searches for the *deepest match* (as depicted in Figure 2) and forwards the message accordingly. There are two advantages over longest-prefix match: the algorithm has the potential to be simpler to implement, and the aggregation does not affect the structure of the routing table (which is just a set of flat names and their associated outgoing port).[14] Instead, all of the aggregation occurs on the naming side, not on the routing side.[15]

How do you know when you can use a concatenation? When naming an object, one thinks about two things: the identifier (which is the name itself) and one or more fetch-terms which we can use to retrieve the object (which, in our case, consist of various concatenated names). These fetch-terms can be included in metadata associated with the name (and signed by the principal of the object) and when asking for the object the request can use these fetch-terms (rather than just the name); the fetch-terms enable the routing system to more easily find the object. It is the responsibility of the principal of an object to not sign any concatenation unless the aggregation invariant holds. This invariant might hold because of administrative relationships (as in DNS names), because of economic relationships (contracts with a CDN), or the organization of a particular piece of content (such as chunks of a large file). This form of aggregation is far more flexible than a strict hierarchy, and several forms of aggregation involving the same object can coexist simultaneously.

This last point is important. Note that in in hierarchies, the object com.CNN.headlines can only fall under the aggregates com and com.CNN. In contrast, with explicit aggregation an object A can fall under an arbitrarily large number of concatenations, say C.B.A and D.F.A.

3.2 Building on Deepest Match

We now discuss several use cases for deepest match.

Structured Content

Consider the concatenation A.B.C, where A represents an aggregate for all of the principal's content, B is the name of a large file belonging to A, and C is an individual chunk in that file. Then, while the network elements will try to forward the fetch towards the chunk (C) if there's a known route, they will fall back to routing towards the aggregate of chunks (B) and eventually to forwarding towards the principal (A), if no more specific routes are known.

In addition, nothing prevents the chunk C to be part of two aggregates B_1 and B_2, in which case the chunk could be fetched using either of the two possible aggregated names (A.B_1.C or A.B_2.C), with the deepest match preferring routing entries towards the individual chunk C or the aggregates B_1 and B_2 before defaulting back to the routes towards A.

Because most content-oriented network architectures rely heavily on caching, a request might well encounter a detailed entry

[12]Simple scaling estimates in [6] suggest that routing on flat names without aggregation is easily within reach of today's technologies, so the entire question of aggregation may be moot. However, for the purposes of this paper we will assume aggregation is necessary.

[13]There is nothing special in self-certifying names that allows this to happen; explicit aggregation can be applied to any naming system. Our only point is that explicit aggregation is more flexible than inherent aggregation.

[14]The deepest match seems intuitively simpler than the longest prefix match, but how much the *implementation* complexities actually differ remains unclear without a detailed comparison of their (hardware and software) implementations. However, since longest prefix matching often transforms into a sequence of exact matching, the differences may be negligible in the end.

[15]See [7] for various anomalies that can occur when routers aggregate entries.

in a routing table (for, say, the individual chunk). Thus, one can limit proactive publishing to higher-level aggregates, but realize the benefits when finer-grain objects are cached nearby.

Content Aggregators

Another form of aggregation arises when third-parties assist in content publishing. With explicit aggregation the principal can use names indicating that it can be found by routing towards a third-party aggregate. We emphasize this sort of outsourcing of content dissemination doesn't imply anything about binding the content to a *topological* location, but only to a third-party service for which the principal might have technical or non-technical reasons to do.[16]

This resembles the use of CDNs today: the principal attaches the name of the content aggregator, say C, to the (beginning) of the name of the object, say A, and then uses this concatenation C.A as a fetch-term. If the principal wants to publish A on its own, in addition to using a content aggregator, it can do so by publishing the object with the fetch-term A. With deepest match, using either fetch-term C.A or A would match routing entries for A; this does not work with longest prefix matching. Moreover, this can be done with multiple third-parties simultaneously, so the principal could use C.A and D.A as fetch-terms, with C and D being different CDNs.

Binding Namespaces

In addition to providing the foundation for route aggregation, the deepest match can assist in mapping between different namespaces.

For instance, assume that we wanted to provide a mapping between a human-readable namespace and a self-certifying namespace. Consider a name "com.CNN.headlines". By representing it as a sequence of hashes, each part representing a longer prefix, we can construct a name A.B.C appropriate for deepest matching. We can make this name self-certifying by using the public key of com as the principal for each of the component names:

A	HASH($com_{public\ key}$) : *
B	HASH($com_{public\ key}$) : HASH(CNN)
C	HASH($com_{public\ key}$) : HASH(CNN.headlines)

The principal of the content could publish the content directly under the name C, and use A.B.C as a fetch-term – and as long as the principal has a certificate from the root principal of the com namespace binding "CNN" to its key, by including the certificate in the object the content principal can continue to use its key to sign the content. This would allow human-readable names to have counterparts in the self-certifying namespace.

The publisher could also opt for a more indirect approach as well: fetching C (either directly, or using the fetch-term A.B.C) could result in a (signed) pointer to a pure self-certifying name without human-readable semantics embedded.[17] As long as the mappings between these two namespaces are relatively static, their dissemination and caching throughout the network could be made more pervasive than the actual content.

[16]Hence, the aggregated name changes only if the used content aggregator is replaced, not if the aggregator's location changes.

[17]An advantage of a pure self-certifying name is that it is semantic free and therefore persistent; for example if the human-readable name points to a home page, and that page changes domains — because the person changed universities — a pure self-certifying name need not change while a human-readable name does; see [14].

4. NAMING AND FLEXIBILITY

No matter what naming scheme is used, external trust mechanisms will be needed to bind keys to RWIs (for self-certifying names) or keys to names (for human-readable). We consider these two cases separately.

A Public-key Infrastructure (PKI) is the most well-known technique for binding keys to names, but has the well-known drawback of requiring universal agreement on the root trust authority and its policies. Decentralized trust mechanisms such as SDSI/SPKI provide an alternative not requiring a universal root of trust. Recall, however, that the network must understand the key–name binding (in order to prevent denial of service), and if the names themselves don't provide this binding then the network must get this binding from the trust mechanism. This precludes the use of a decentralized solution without a global root (as it would be untenable to have different portions of the network invoking different bindings), and forces the use of a PKI. Thus, human-readable names require the use of a PKI.

Not only do human-readable names reduce the flexibility in trust mechanisms, their name–RWI binding is imperfect (*e.g.*, which RWI does the name icsi.org refer to?), so an external mechanism must be used to provide more secure bindings between names and RWIs (in addition to the PKI used to bind keys to names).

Any binding to RWIs (whether key–RWI or name–RWI) involves human-level notions of identity so the binding cannot be completely reduced to cryptographic mechanisms. In particular, we imagine that search engines and social networks will play an important role in future trust mechanisms involving RWIs. In fact, the whole purpose of search engines is to map real-world identities (based on given keywords) to their relevant network names (URLs).[18]

Social networking platforms have the opportunity to provide trust on a more personal level. The networks of friendship could map directly to the use of webs of trust, facilitating decentralized trust management: in this model, establishing a trust in a name becomes a matter of traversing your social network and finding the certificate chain(s) from yourself to the name via your friends.

These and other RWI-based trust mechanisms are imperfect, but they are the best we have, regardless of what naming scheme is used. Moreover, the diversity of these mechanisms requires that they be completely separate from the network.

Self-certifying names provide a clean, algorithmic binding between names and keys that is understandable by the network, and leave the more amorphous RWI bindings to a collection of trust mechanisms external to the network that can evolve over time as new sources of trust arise (*e.g.*, search engines and social networks are relatively recent developments, and they will undoubtedly be augmented by future innovations).

Human-readable names provide an imperfect binding between names and RWIs, and thus require external mechanisms for both bindings; moreover, they dictate the use of a PKI-like infrastructure for the network because the network must understand the key–name bindings.

5. DISCUSSION

The intuition underlying this paper is summarized by the following design maxim: *You should architect for security and flexibility*

[18]Having the search engine to return a key (and/or self-certifying name) for given a real-word identity description would be straightforward, and at least one large search engine company is planning to do so.

but engineer for performance. We now review the findings of this paper with this maxim in mind.

There are essentially three options for the names used to route requests:

Intrinsically bound to RWIs but not to keys. In this case, to prevent denial of service the network must be able to determine the keys itself, which means that the network must understand the trust mechanism for key–name binding. This leads to the use of a PKI, which reduces flexibility in trust management.

Intrinsically bound to both keys and RWIs. In this subcategory of self-certifying names (motivated by the need to include keys when fetching content to prevent denial of service), the self-certifying component binds the key to the name and human-readable component binds the RWI to the name. While superficially attractive, this approach is suboptimal because the name–RWI binding provided by human-readable names is not reliable. This would then leave a security hole unless replaced or augmented by an external mechanism for name–RWI binding, which brings us to the next option.

Intrinsically bound only to keys. This is the canonical self-certifying naming paradigm, where key is bound to name through self-certification and the RWI–key binding is provided externally through a variety of mechanisms. While the RWI–key binding will never be perfect, at least it can continue to evolve and improve. This approach provides maximal flexibility in trust mechanisms, cryptographically secure name–key binding, and still allows one (though various forms of aggregation) to engineer for better performance.

This is our rationale for promoting the use of self-certifying names. We view this not as the final word on the subject, but hopefully the beginning of a reasonable exchange of viewpoints that can help lead the community to a deeper understanding of the issues.

Acknowledgments

This work was supported by TEKES as part of the Future Internet program of TIVIT (Finnish Strategic Centre for Science, Technology and Innovation in the field of ICT).

6. REFERENCES

[1] B. Ahlgren et al. Second NetInf Architecture Description. Technical Report D-6.2 v2.0, 4WARD EU FP7 Project, 2010.

[2] C. Dannewitz, J. Golić, B. Ohlman, and B. Ahlgren. Secure Naming for a Network of Information. In *Proc. of Global Internet Symposium*, March 2010.

[3] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylönen. SPKI Certificate Theory. RFC 2693, IETF, September 1999.

[4] M. J. Freedman, M. Arye, P. Gopalan, S. Y. Ko, E. Nordstrom, J. Rexford, and D. Shue. Service-Centric Networking with SCAFFOLD. Technical Report TR-885-10, Princeton University, CS, September 2010.

[5] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking Named Content. In *Proc. of CoNEXT*, December 2009.

[6] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica. A Data-Oriented (and Beyond) Network Architecture. In *Proc. of SIGCOMM*, August 2007.

[7] A. R. R. White, D. Slice. *Optimal Routing Design*. Cisco Press, 2005.

[8] J. Rajahalme, M. Särelä, K. Visala, and J. Riihijärvi. On Name-based Inter-domain Routing. *Comput. Netw.*, December 2010.

[9] R. L. Rivest and B. Lampson. SDSI – A Simple Distributed Security Infrastructure. Technical report, MIT, 1996.

[10] D. Smetters and V. Jacobson. Securing Network Content. Technical report, PARC, October 2009.

[11] C. Soghoian and S. Stamm. Certified Lies: Detecting and Defeating Government Interception Attacks Against SSL. In *Proc. of HotPETS*, July 2010.

[12] D. Trossen. On long-lived routing identifiers. http://www.fp7-pursuit.eu/PursuitWeb/?p=244, October 2010.

[13] D. Trossen, M. Särelä, and K. Sollins. Arguments for an Information-centric Internetworking Architecture. *SIGCOMM CCR*, 40, April 2010.

[14] M. Walfish, H. Balakrishnan, and S. Shenker. Untangling the Web from DNS. In *Proc. of NSDI*, March 2004.