Schematizing Trust in Named Data Networking

Yingdi Yu
UCLA
yingdi@cs.ucla.edu

Alexander Afanasyev
UCLA
afanasev@cs.ucla.edu

David Clark
MIT
ddc@csail.mit.edu

kc claffy
CAIDA
kc@caida.org

Van Jacobson
UCLA
vanj@cs.ucla.edu

Lixia Zhang
UCLA
lixia@cs.ucla.edu

ABSTRACT

Securing communication in network applications involves many complex tasks that can be daunting even for security experts. The Named Data Networking (NDN) architecture builds data authentication into the network layer by requiring all applications to sign and authenticate every data packet. To make this authentication usable, the decision about which keys can sign which data and the procedure of signature verification need to be automated. This paper explores the ability of NDN to enable such automation through the use of *trust schemas*. Trust schemas can provide data consumers an automatic way to discover which keys to use to authenticate individual data packets, and provide data producers an automatic decision process about which keys to use to sign data packets and, if keys are missing, how to create keys while ensuring that they are used only within a narrowly defined scope ("the least privilege principle"). We have developed a set of trust schemas for several prototype NDN applications with different trust models of varying complexity. Our experience suggests that this approach has the potential of being generally applicable to a wide range of NDN applications.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*; D.4.6 [**Software**]: Security and Protection

Keywords

Security, Named Data Networking

1. INTRODUCTION

Designing secure systems and network applications involves properly authenticating multiple entities in the system and granting these entities with the minimum set of privileges necessary to perform operations. In contrast to traditional IP networks where applications usually rely on an additional layer (e.g., Transport Layer Security [9]) to authenticate connections, Named Data Networking (NDN) [18, 20] is a proposed data-centric Internet architecture that requires every application to name and sign the produced network-level data packets and to authenticate received packets. To utilize the data-centric security of NDN without requiring application developers and users to be security experts, system-level support is needed to automate the process of packet signing and authentication.

The power of the NDN architecture comes from naming data hierarchically with the granularity of network-level packets and sealing named data with public key signatures. Producers use key names to indicate which public key a consumer should retrieve to verify signatures of produced data packets. In addition to fetching the specified keys and performing signature verification, consumers also match data and key names to determine whether the key is authorized to sign each specific data packet.

To facilitate this matching process, we introduce the concepts of *trust rules* and *trust schemas*. A set of trust rules defines a trust schema that instantiates an overall trust model of an application, i.e., what is (are) legitimate key(s) for each data packet that the application produces or consumes. The fundamental idea is that each trust rule defines a relationship between the name of each piece of data and its signing key, e.g., both must share the same prefix, share the same suffix, and/or have specific name components at certain position of the names. Given a trust schema that correctly reflects the trust model of the application, data producers can select (and if necessary generate) the right keys to sign the produced data automatically, and consumers can properly authenticate each retrieved data packet.

In this paper we describe how NDN naming and the use of *trust schemas* enable automation of data signing and authentication in NDN applications with complex trust models. We have implemented a prototype of a trust schema in NDN application development libraries (ndn-cxx and NDN-CCL) which have been used to power the trust management of several NDN applications, including the NDN Forwarding Daemon (NFD), NDN Link State Routing Protocol (NLSR), NDN Domain Name System (NDNS), NDN Repository System (repo-ng), and ChronoChat applications [15].

We can summarize our contributions as three-fold. First, we identified a *name-based trust management* mechanism, which we hope not only can help secure NDN applications, but can also benefit other data-centric systems. Second, we invented *trust schema* as a systematic way to define application trust models and introduce the concept of *security*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ICN'15, September 30–October 2, 2015, San Francisco, CA, USA.
© 2015 ACM. ISBN 978-1-4503-3855-4/15/09 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810156.2810170.

design pattern to facilitate application development. Third, we developed a prototype of *trust schema interpreter* that has been successfully tested to automate authentication and signing process in a set of diverse NDN applications.

We organize the rest of this paper as follows. Section 2 introduces data authentication in NDN and its threat model. We then explain the value of a trust schema (Section 3), our trust schema design (Section 4), its use in automating trust management (Section 5), and other considerations in their use (Section 6). Finally we review related work and summarize our contribution.

2. DATA AUTHENTICATION IN NDN

NDN fosters a data-centric security model at the network layer [18]: each data packet is uniquely named and this name is bound to the content using a signature. Besides being a transport unit as an IP packet, an NDN data packet is also a storage unit, so that the signature carried in a data packet secures its content whether the packet is in motion or in storage. Beside the name, content, signature, and a few other fields, a data packet also contains a `KeyLocator` field [16] to indicate the name of the public key that is used to produce the signature. In NDN, a key is simply another piece of named NDN content (Figure 1). Like any other data packet, a packet carrying a public key is also signed, making it equivalent to a certificate [19]. Because NDN names the content carried in a packet, for simplicity, we will use the term “key” to refer to an NDN data packet that carries a public key.

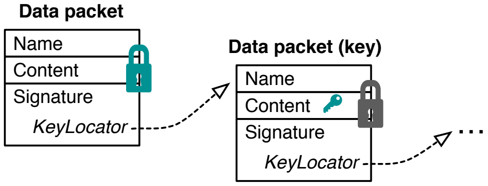


Figure 1: Authentication elements in NDN data packet

In the rest of the paper we assume that given a name, the network can directly retrieve the corresponding data packet. Other considerations, including fetching data packets whose names are not globally routed [1], may require additional steps in data retrieval but will not affect the security model described in this paper.

2.1 Example of Data Authentication

We use a simplified blog website framework as an example throughout the paper to illustrate a possible trust model and our proposed approach to schematize it. The framework includes four groups of entities (Figure 2): the website, website administrators, blog authors, and articles. The website may have a few administrators, who can authorize authors to publish articles on the website. Trust relations between these entities in NDN terms can be captured by signed data packets and chains of keys. When an administrator installs the website software, the installer generates a key[1] to act as the root of trust for the website. The installer process also creates a key for the initial administrator and signs it with the website's key. The initial administrator can further delegate management privileges to other administrators by signing their keys, and any administrator can add authors into the system by signing the authors' keys. Each author can publish on the website by signing the produced articles using a valid author key.

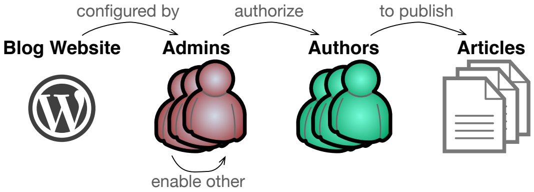


Figure 2: Entities of a simple blog website framework

When a reader retrieves an article, he or she can recursively follow the `KeyLocator` field in each data packet to retrieve the key of the author who wrote the article, the key of the administrator who authorized the author, and the key of the blog website where the article is published. If the reader accepts the website trust model and trusts the public key of the website (or uses PKI or web-of-trust mechanisms to verify authenticity of the key), the reader can reliably authenticate legitimate articles through a sequence of data packet signature verifications.

2.2 Threat Model

Threats to data authentication integrity in NDN include failed authentication, mis-authentication, and key compromise. Failed authentication of a legitimate key (false negative) can result in a consumer treating valid data as malicious, potentially leading to denial of service. Mis-authentication of a mis-configured or malicious key (false positive) can cause consumers to accept false data. These errors can occur when the trust schema (data-key relations) is incorrectly or unclearly defined, or when the authentication mechanism does not fully adhere to the defined schema. A set of commonly used trust schemas written by security experts not only can mitigate these threats, but also facilitate automation of both signing and authentication mechanisms.

When a legitimate key is compromised, an attacker can obtain privileges associated with this key. To mitigate this threat we enforce “the least privilege principle”: each key must have a restricted non-elevating usage scope to limit the damage upon key compromise, and keys with broader privileges should be used as infrequently as possible.

3. WHY WE NEED A TRUST SCHEMA

In general, the relationship between data and key names can be complex. Depending on an application's naming structure and trust model, data authentication may involve a chain of keys (*authentication path*) across several different namespaces. We use our blog website example introduced in Section 2 to illustrate the necessity of authentication across different namespaces, and highlight the need for the trust schema to concisely express complex trust model relations.

The blog website framework defines entities in the system and also their trust relationships. Since everything is explicitly named in NDN, the framework also needs to define a naming representation of the entities. Figure 3 shows a possible representation: assuming the website owns “`/a`” namespace and allocates “`/a/blog`” to blog publishing, ar-

[1] This key may be self-signed or later secured using some trust model, e.g., PKI or web-of-trust.

ticles are represented as data packets under the "`/a/blog/article`" namespace, with category, publication year, and unique article identifier; each author obtains a key under the "`/a/blog/author`" namespace with an author identifier;[2] each administrator obtains a key under the "`/a/blog/admin`" namespace with an administrator identifier; and the website itself has a configuration key with the name "`/a/blog`" (e.g., created during the installation of the blog). An implementation of this blog website framework must capture the trust relationship between all these entities in terms of the relationship between NDN namespaces. However, this comprehensive naming structure leads to the fact that an authentication path following the trust model may need to traverse three namespaces: "`/a/blog/article`", "`/a/blog/author`", and "`/a/blog/admin`" as shown in Figure 3.

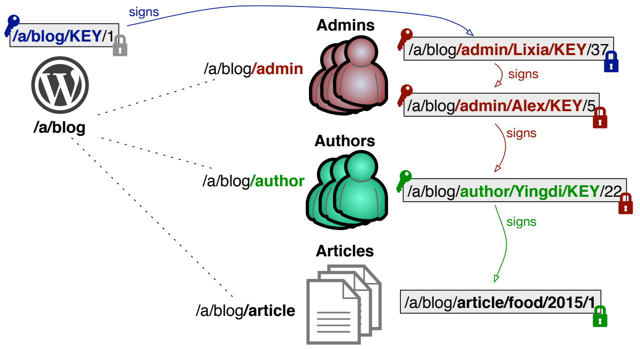


Figure 3: Example of namespaces and authentication paths in a blog website "`/a/blog`"

In theory, it is possible for application developers to hardcode all relationships in the trust model, i.e., relationships between articles and authors keys, between authors and administrators keys, between administrators keys and other administrators keys, and between administrators keys and the configuration key of the website. However in practice, even with a simple trust relationships as in our example, this process is non-trivial and error-prone. A small implementation error may compromise the security of the entire website. For example, a website implementation that accidentally associates author management with author keys rather than with administrator keys may allow authors to authorize another author without the permission from an administrator. Or, an article-publishing application that mistakenly uses an administrator key to directly sign an article violates the least privilege principle, and may also prevent browsers that comply with the trust model from authenticating articles.

In contrast, when the trust relationships are captured by a set of well-defined rules that match data and key names (*trust schema*), a system-level tool interpreting these rules can automatically execute authentication and signing procedures. This ability to automate unburdens developers from individually handling sophisticated data signing and authentication. A trust schema also makes it feasible for security experts to define a set of generalized trust models (e.g., one for blog websites, one for mail services, etc.) that other application instances of the same type can reuse. Each reuse can continue to refine and debug the schema, improving it for future applications.

4. TRUST SCHEMA

In this section, we present the trust schema as a tool to define trust models in a generalized way. A trust schema comprises a set of linked trust rules and one or more trust anchors. As we will show later in this section, the trust schema mechanism can be used to automate both authentication and signing processes. To define trust schema rules, we will use a notation similar to regular expressions to express the *name pattern*. Table 1 gives a brief summary of the syntax elements we use in name patterns that are formally defined in [17].

Table 1: Elements of name patterns used in trust schema definitions

`<name>`	Match name component `name`
`<>`	Match any single name component, i.e., wildcard
`<name><>`	Match name component `name` followed by any single name component
`<>*`	Match any sequence of name components
`(...)`	Match pattern inside the brakets and assign it as an indexed sub-pattern
`\n`	Reference to the n-th indexed sub-pattern
`[func]`	Match (for authentication) or specialize (for signing) name component according to function `func` defined pattern, i.e., wildcard specializer
`rule(arg1, ...)`	Derive a more specific name pattern from `rule`'s data name pattern with arguments `arg1`, ...

4.1 Trust Rule

A trust rule is an association of the data name with its corresponding signing key name. There are multiple ways to represent such association. For example, Figure 4(a) shows a simple direct association between an article name and its corresponding author name. This rule precisely captures that the article "`.../food/2015/1`" must be signed by author key "`.../Yingdi/KEY/22`", but says nothing about other articles or authors, even those that share the same naming patterns. If we can generalize the name relationships in trust rules, and reliably link rules to one another, we can construct concise, sophisticated, robust, and re-usable trust models.

4.1.1 Generalizing Trust Rules

A well-defined trust model usually associates the same type of data with the same type of keys, e.g., articles should always be signed by the authors. We can use the naming structure of a given application (or a set of applications that share the same naming structure) to create a set of rules to define the relationships between name patterns for data and keys in that application. This set of trust rules then captures the complete trust model for the application.

In the blog example, all articles share the same prefix "`/a/blog/article`", but each article has its own category, year, and article identifier. One way to generalize this relationship is to use name patterns as shown in Figure 4(b). In Figure 4 and later examples, we use the wildcard "`<>`" to match any name component (i.e., the schema does not impose any restrictions on the content of the name component), "`[user]`" to match alphanumerical user identifiers, and "`[id]`" to match numerical key identifiers.

[2] The last two components of each key name are "`KEY`" and a key identifier. This naming convention allows authors to change keys over time.

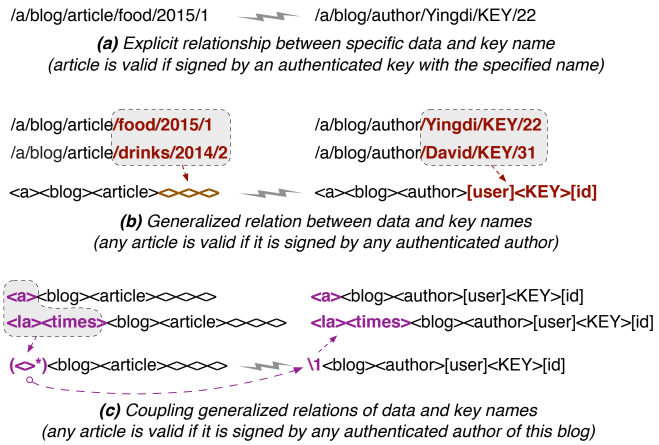
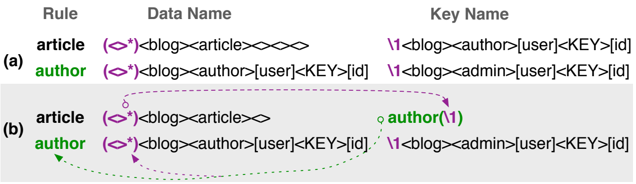

/a/blog/article/food/2015/1
/a/blog/author/Yingdi/KEY/22
(a) Explicit relationship between specific data and key name
(article is valid if signed by an authenticated key with the specified name)
/a/blog/article/food/2015/1
/a/blog/article/drinks/2014/2
/a/blog/author/Yingdi/KEY/22
/a/blog/author/David/KEY/31
<a><blog><article><><><>
<a><blog><author>[user]<KEY>[id]
(b) Generalized relation between data and key names
(any article is valid if it is signed by any authenticated author)
<a><blog><article><><><><>
<la><times><blog><article><><><><>
<a><blog><author>[user]<KEY>[id]
<la><times><blog><author>[user]<KEY>[id]
(<>*)<blog><article><><><><>
\1<blog><author>[user]<KEY>[id]
(c) Coupling generalized relations of data and key names
(any article is valid if it is signed by any authenticated author of this blog)

Figure 4: Trust rule generalization



	Rule	Data Name	Key Name
(a)	article	(<>*)<blog><article><><><><>	\1<blog><author>[user]<KEY>[id]
	author	(<>*)<blog><author>[user]<KEY>[id]	\1<blog><admin>[user]<KEY>[id]
(b)	article	(<>*)<blog><article><>	author(\1)
	author	(<>*)<blog><author>[user]<KEY>[id]	\1<blog><admin>[user]<KEY>[id]

Figure 5: Generalization of trust rule linkage: (a) implicit linkage; (b) explicit linkage

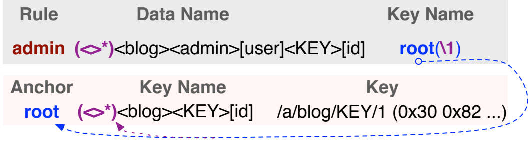
In general, trust models must explicitly associate a data name with its signing key name through matching of name components. In our example, both the article name and the author name must share the same website name ("/a"). To capture this constraint, we leverage sub-patterns and repetition syntax, as highlighted in Figure 4(c). We believe this syntax is sufficiently general to capture complex trust model frameworks, allowing reuse of trust models by different application instances. In other words, the trust schema for our blog example can be used by any other blog website that shares the same trust model.

4.1.2 Linking Trust Rules

A trust model should also properly associate keys with their signing keys, to ensure that a data consumer can reliably construct chains of keys to authenticate data and that a data producer can correctly choose or initialize its signing keys.

Figure 5(a) defines "article" and "author" trust rules. The key name pattern in the "article" rule will always match the data name pattern of the "author" rule, therefore both rules are implicitly linked. However, in order to ensure integrity of the trust model, the schema should unambiguously describe an authentication path (or paths) for each valid data packet. Therefore, each rule has to be explicitly linked to other rule(s) in the trust schema definition.

To explicitly link rules, we assign each rule a unique identifier to be used in a function-like way as part of the key name pattern, as shown on Figure 5(b). In other words, invoking such rules is similar to invoking a function: invocation substitutes the key name pattern with the data name pattern from the invoked rule, specializing it with the supplied patterns or references to the indexed sub-patterns. In our example, the "article" rule invokes the "author" rule passing to it the first indexed sub-pattern. For the "/a/blog/article/food/2015/1" article, the sub-pattern will expand to "/a" and the invocation to the "author" rule will return "<a><blog><author>[user]<KEY>[id]" name pattern. This linkage imposes the restriction that only authorized authors of blog "/a/blog" can sign and publish articles of the blog.



Rule	Data Name	Key Name
admin	(<>*)<blog><admin>[user]<KEY>[id]	root(\1)

Anchor	Key Name	Key
root	(<>*)<blog><KEY>[id]	/a/blog/KEY/1 (0x30 0x82 ...)

Figure 6: Example of linking trust rule and anchor

4.2 Trust Anchor

To be complete, a trust schema must also include one or more trust anchors which serve as bootstrapping points for the trust model. A trust anchor is a key that is pre-authenticated using an out-of-band mechanism, e.g., manually installed or comes with software packages. In the trust schema we express trust anchors as special rules that include a key name pattern and a pre-authenticated key. Every successful authentication path must end at a trust anchor. Therefore, a trust schema must always include a way for trust rules to establish the link(s) from data or key names down to a trust anchor. Figure 6 shows an example of the trust rule "admin" linking to the trust anchor "root".

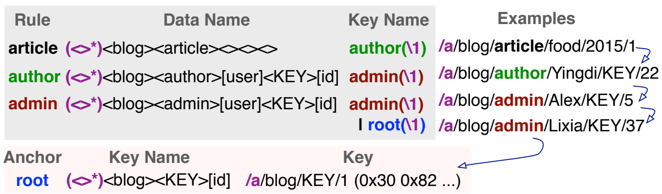
The trust anchor performs two important functions. First, it explicitly defines not only the name of the trust anchor, but also the key bits, i.e., if a packet is signed with a key that matches the name pattern in a trust anchor, this packet must be authenticated using the pre-specified key bits. Second, the anchor explicitly restricts the privilege of the pre-authenticated public key using name pattern, so that the key cannot be used to authorize anything else. For example, an administrator's key of another website "/another/blog/admin/Carl" will not be a valid administrator's key for "/a/blog": the expanded key pattern "<another><blog><KEY>[id]" will not match the blog's trust anchor "/a/blog/KEY/1". Note that the schema also prohibits another website's administrator key to be signed with the blog's trust anchor: the "admin" rule will rightfully reject such a key.

4.3 Crypto Requirements

In addition to providing a generalized formal definition of trust rules and trust anchors, a trust schema must also include cryptographic requirements on data signatures, such as the hash and signing algorithm and the minimum key size. These requirements are not directly related to naming, but can help prevent consumers from accepting data with easily compromised signatures. Therefore, a trust schema should clearly state these parameters as an essential part of a trust model.

4.4 Trust Schema Examples

We now demonstrate how the trust schema we described so far can express two different trust models. The first trust model is for our blog website framework, and the second is an example of a model that resembles the trust model of



Rule	Data Name	Key Name	Examples
article	(<>*)<blog><article><><><>	author(\1)	/a/blog/article/food/2015/1
author	(<>*)<blog><author>[user]<KEY>[id]	admin(\1)	/a/blog/author/Yingdi/KEY/22
admin	(<>*)<blog><admin>[user]<KEY>[id]	admin(\1)	/a/blog/admin/Alex/KEY/5
		I root(\1)	/a/blog/admin/Lixia/KEY/37

Anchor	Key Name	Key
root	(<>*)<blog><KEY>[id]	/a/blog/KEY/1 (0x30 0x82 ...)

Figure 7: Trust schema the blog website framework with "/a/KEY/1" as the trust anchor

DNSSEC and *strictly* follows the naming hierarchy to match data and key names.

4.4.1 Blog Website Framework

In the blog website example, the trust rules must capture the relationship between articles and authors, between authors and administrators, as well as between administrators and blog website configuration (the blog's trust anchor). An example of the trust schema that can achieve these goals is shown in Figure 7. Note that this schema assumes that the blog's configuration key "/a/blog/KEY/1" is pre-authenticated (i.e., a trust anchor). Depending on the specific usage scenario, a blog reader may further authenticate the configuration key using a hierarchical trust model similar to the example in Section 4.4.2, or using some other trust model, e.g., web-of-trust.

The first rule in the example schema, "**article**", captures the trust constraint that authors must sign their articles with their keys. Similarly, the "**author**" rule ensures that only blog administrators can sign authors' keys. The final "**admin**" rule defines two possible relations for administrators' keys in the security framework: (1) existing administrators may delegate administrator privileges to another person; and (2) authentication paths for the administrator keys must terminate at the blog website trust anchor.

Note that although every trust rule in the trust schema in Figure 7 uses the repeated wildcard "<>*" to match the website prefix, the prefix is always determined (specialized) at the moment when the "**article**" rule captures the original article data name. After the "**article**" rule captures "/a/blog/article/food/2015/1" data, prefix "/a" is propagated to the "**author**" rule as a reference to the first subpattern, then to the "**admin**" rule, and down to the "**root**" trust anchor.

4.4.2 Hierarchical Trust Model

In a linear hierarchical trust model, with DNSSEC [2] as a prominent example, a single rule can capture the relationship between all the data and key names; in plain English, this rule is "the signing key name must be a prefix of the data name." Because key names should be unique and need to include additional suffix components as shown on Figure 8, the trust schema for the hierarchical relationship in NDN needs to consider these additional components.[3] The overall trust in this model can be bootstrapped using one or more trust anchors associated with the top level namespace(s).

Figure 9 shows an example of the trust schema that defines the hierarchical trust relationships, consisting of a single rule

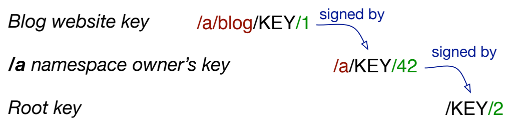
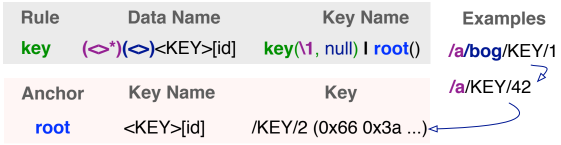


Figure 8: Example of naming in hierarchical trust model



Rule	Data Name	Key Name	Examples
key	(<>*)(<>)<KEY>[id]	key(\1, null) I root()	/a/bog/KEY/1
			/a/KEY/42

Anchor	Key Name	Key
root	<KEY>[id]	/KEY/2 (0x66 0x3a ...)

Figure 9: Trust schema for the hierarchical trust model with "/KEY/2" as the trust anchor

and a trust anchor. The rule "**key**" captures that keys at each level of the hierarchy must be signed by the keys from the parent namespace, i.e., the prefix before "KEY" of the signing key name must be one component shorter than the name of the key itself. The trust anchor ensures that the authentication path discovery terminates when it reaches the root namespace: when the prefix of the signing key before "KEY" is empty (just "/"), then it must be signed by the specified "/KEY/2" key.

The "**key**" rule is recursively linked to itself and to the trust anchor. In these cases, when matching data and key names, all specified patterns need to be considered, with anchor rules taking precedence. For a key "/a/blog/KEY/1", the rule "**key**" will extract the parent namespace of the key (i.e., "/a") and derive two name patterns: "<a><KEY>[id]" and "<KEY><2>". Given the signing key name matches the first pattern, the process recursively continues with the same rule, until there is a match with the trust anchor.

If the key's KeyLocator does not match any key name pattern, it implies that the key does not comply with the trust model and should be treated as an invalid key.

4.5 Schema for Authentication

For each data packet, the trust schema determines a valid authentication path(s) within the corresponding trust model. Given that the trust schema is expressed as formally defined rules, an *authentication interpreter* of the trust schema can automate the whole authentication process for any given trust model (Section 5.1).

For each received data packet, the authenticating interpreter finds the corresponding trust rule by matching the name of the packet against the specified name patterns in the rules. If the packet and its KeyLocator comply with constraints of the found trust rule, the interpreter can then retrieve the public key according to the data's KeyLocator and recursively inspect the retrieved key according to the trust schema, until reaching a trust anchor or a pre-defined limit on the number of recursive steps. In the former case, the interpreter has collected all the intermediate public keys on a valid authentication path, thus can verify signatures starting from the trust anchor up to the received data packet. When the interpreter cannot find a rule that matches the received data packet, or the constructed authentication path loops, or the path becomes overly long, the interpreter declares failure to discover the authentication path.

[3] For simplicity, in this example we consider only authentication of DNS keys, but the trust model and schema can be easily extended to other DNS data, as shown with the blog website example.

The received data packet is authenticated only if there is a valid authentication path according to the trust schema, and each signature on the path is verifiable and satisfies the cryptographic requirements of the schema. In other words, either failure to discover authentication path or failure to verify any signature on the authentication path implies that the received data packet cannot be authenticated with the interpreted trust model.

4.6 Schema for Signing

One can also view the trust schema as a collection of constraints on a data packet's signing key, with respect to its name, signature, key type and size, etc. Thus, the trust schema also specifies the required signing process, i.e., how to select or generate signing keys given the name of the data packet. Effectively, this allows automation of the signing process using a *signing interpreter* of the trust schema (Section 5.2).

The signing interpreter takes a data packet as an input and looks up the corresponding trust rule. Instead of checking for compliance of the data's name and `KeyLocator` to the trust rule, it infers the correct name of the key to be used to sign the data packet. If this key exists on the system, the interpreter will immediately sign and return the data packet. If the key does not exist, the interpreter will try to generate the key with the specified name and crypto requirements, and then sign this key by recursively re-interpreting the same schema again with the generated key as a new input. See further details in Section 5.2 on how the interpreter can generate key names based on rules in the trust schema.

Note that it is not always possible for the interpreter to automatically generate all necessary keys, without out-of-band verification mechanisms. For example, if a not-yet authorized author is trying to sign an article for publication, the interpreter will fail to sign it, as the author does not have a valid key to sign an article, nor a key to endorse an author on the blog, nor a key to configure a new administrator in the system. Even in this case, the interpreter can still generate useful diagnostic information, e.g., which keys are missing and how to obtain them.

5. AUTOMATING TRUST

Now that we have introduced the concept of schema-based data authentication and signing, we will describe in detail how to automate these processes, using the blog website framework as an example.

5.1 Automating Authentication

Each step of the authentication path for data (key) packets is defined by the rules of the trust schema. Rules are linked together through a function-like invocation of rule names as part of the key name pattern definition, as shown in Figure 7. The authentication process moves forward (from one step to the next) only if the data (or key) satisfies the conditions of the rule. We can model this authentication process as a Finite State Machine (FSM), with each state representing a rule and state transitions representing function-like invocations. This way, once a data packet enters the FSM, the FSM's states define the packet's authentication path, and an automatic process can walk through these states until exiting the FSM with success or failure.

Execution of the FSM processing requires a trust schema interpreter. The interpreter used for data authentication, which we call *authenticating interpreter*, takes data packets as input, requests public keys when necessary, and outputs whether the received packet is authenticated or not. Given the trust schema for a trust model, an authenticating interpreter can effectively automate the process of data authentication for this trust model. Figure 10 shows the FSM of an authenticating interpreter for the blog website trust model discussed in Section 4.4.1.

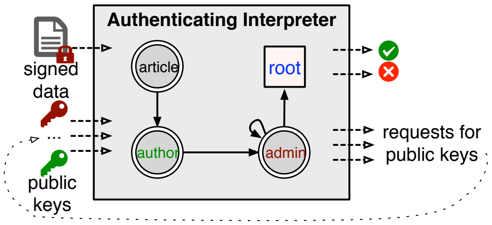


Figure 10: Finite state machine for the authentication interpreter of the blog website trust model schema

5.1.1 Authentication State

Whenever a new data packet arrives at the FSM, the interpreter determines the corresponding initial state by checking the data name against the name patterns for each state. After that, the interpreter initiates the key name checking procedure, including steps to:

- extract components from the data name according to the defined sub-patterns;
- derive the key name pattern from the rule's key name functions with the extracted components;
- check if the data's `KeyLocator` matches the derived key name pattern.

If the data packet passes the key name checking, the authentication process transitions to the downstream state of the FSM: the interpreter requests the key identified by the `KeyLocator` field carried in this packet and pauses FSM processing until the key is retrieved. When the key is delivered to the interpreter, the interpreter initiates a new instance of the same checking procedure at the state on which the FSM processing previously paused. Whenever the FSM transitions to a trust anchor state, the interpreter immediately triggers verification of signatures, following the reverse path of transitions in the FSM.

5.1.2 Walking Through the State Machine

In this section we demonstrate how the authentication automation can work for the blog website trust model. We use an article data packet with name "`/a/blog/article/food/2015/1`" signed by an author key "`/a/blog/author/Yingdi/KEY/22`" as an example to show how the authentication process goes through the state machine shown in Figure 10.

Initial state.

Based on the trust schema, the article name "`/a/blog/article/food/2015/1`" will be captured by the "**`article`**" rule, thus the authentication process starts from the corresponding "**`article`**" state. When executing the key name checking procedure, the interpreter will extract "**`<a>`**" as the first sub-pattern and use it to derive a key name pattern through a function-like invocation of the "author" rule. The resulting pattern "<a><blog><author>[user]<KEY>[id]" will successfully match the KeyLocator field of the data packet and the FSM will transition to the downstream "author" state.

State transition.

At this point, the interpreter makes a request for "/a/blog/author/Yingdi/KEY/22" key and pauses processing until the key is retrieved. After retrieving the requested key, the interpreter resumes operations at the "author" state with the retrieved key as an input. Similarly, the interpreter extracts "<a>" as the first sub-pattern from the author key name and derives through the "admin" rule a key name pattern "<a><blog><admin>[user]<KEY>[id]". Assuming that the retrieved key is signed with an admin key "/a/blog/admin/Alex/KEY/5", the FSM will transition to the corresponding "admin" state.

Self-loop transition.

The "admin" rule in the website trust schema links to two trust rules, of which one is the "admin" rule itself. This self-linked rule represents a management privilege delegation from one administrator to another administrator and is represented by a self-loop transition in the FSM. This transition can capture an administrator key "/a/blog/admin/Alex/KEY/5" signed with another administrator key "/a/blog/admin/Lixia/KEY/37". In this case, the FSM transitions to the same "admin" state over the loopback link and the interpreter requests for the other administrator key and pauses the FSM processing again.

Note that a self-loop transition can potentially accept authentication paths that contain loops or excessively long authentication paths. To prevent these loops, the interpreter can record names of every intermediate key that each state has observed during the authentication process, and abort processing when detecting a duplicate. To prevent excessively long authentication paths, e.g., from a carefully crafted key chains in attempts to cause denial of service attacks, the interpreter should set a limit on the number of state transitions.

Transitioning towards the trust anchor state.

When the interpreter retrieves the public key "/a/blog/admin/Lixia/KEY/37", it can repeat the key name checking procedure on the "admin" again, deriving two patterns for key name matching: "<a><blog><admin>[user]<KEY>[id]" (from the "admin" rule) and "<a><blog><KEY>[id]" (from the trust anchor "root"). If "/a/blog/admin/Lixia/KEY/37" key was signed by "/a/blog/KEY/1" (the specified trust anchor), the second name pattern would match the KeyLocator. In this case, the process immediately transitions to the trust anchor state, triggering initiation of the signature verification procedure.

Signature verification.

Once the signature verification procedure is triggered, the interpreter will follow the reverse path of FSM back to the original data packet, terminating with failure if at any step it cannot verify the signature. In the example, the process will start with validating "/a/blog/admin/Lixia/KEY/37" key using the trust anchor key, following checking signature of "/a/blog/admin/Alex/KEY/5" using the validated admin key, similarly for the author key "/a/blog/author/Yingdi/KEY/22", terminating with checking signature of the received article data packet using validated author key.

5.2 Automating Signing

Another version of the trust schema interpreter, a *signing interpreter*, can use a trust schema to automate selection of signing keys and generation of keys when necessary/possible. Similar to the authenticating interpreter, the signing interpreter compiles a trust schema to an FSM (Figure 11), but processes an *unsigned* data packet as input and outputs the data packet signed with a key that conforms to the trust model (or fails). During processing, the interpreter interacts with the private key store (e.g., Trusted Platform Module, TPM) to request data signing and create signing keys when they are not yet available.[4]

The signing interpreter will fail to sign the supplied data packet if the required key is not available in the local TPM and when the key generation procedure has to cross security boundaries, e.g., a remote admin needs to sign new author's key. In such cases, additional out-of-band mechanisms are necessary to generate proper keys. For example, when a blog author is not an admin of the blog, he or she needs to call or email one of the administrators to give him or her the permission to publish articles. While it is impossible for the signing interpeter to generate keys completely automatically, it can provide assistance in creating the required keys (e.g., generate signing requests) and simplify complex cryptographic operations.

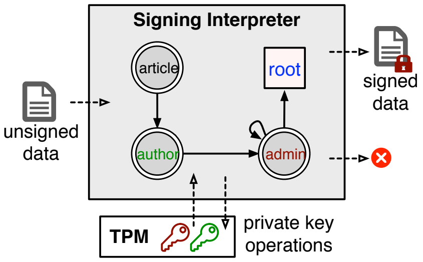
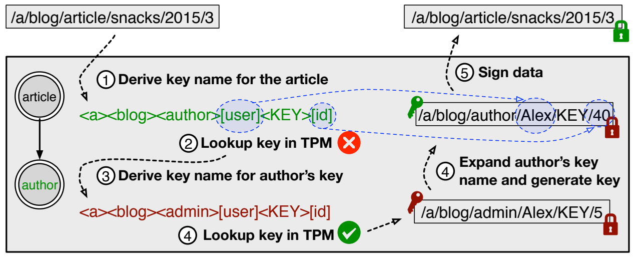


Figure 11: Signing interpreter for the blog website trust model schema

5.2.1 Key Selection

Given a data packet, the signing interpreter can derive the name pattern of a key that is allowed to sign this data according to the trust model. For this purpose, it finds the state in the FSM that corresponds to the data packet, and expands the corresponding signing key name pattern. For example, let us assume that an administrator of the blog wants to publish his article "/a/blog/article/snacks/2015/3". This data packet will enter the FSM from the "article" state, at which point the interpreter can derive the key name pattern "<a><blog><author>[user]<KEY>[id]", as shown in step 1 in Figure 12. With the derived name pattern and the crypto requirements from the trust schema, the interpreter will search a qualified key in the TPM (step 2 in Figure 12). In our example, the admin is publishing a blog article for the

[4] Ideally, a signing interpreter should be implemented as a trusted service provided by operating system.


/a/blog/article/snacks/2015/3
/a/blog/article/snacks/2015/3
article
author
1 Derive key name for the article
5 Sign data
<a><blog><author>[user]<KEY>[id]
/a/blog/author/Alex/KEY/40
2 Lookup key in TPM
4 Expand author's key name and generate key
3 Derive key name for author's key
<a><blog><admin>[user]<KEY>[id]
/a/blog/admin/Alex/KEY/5
4 Lookup key in TPM

[user] is a function to expand to a local user name
[id] is a function to expand to a unique numerical identifier

Figure 12: An interpreter processing the blog website trust schema directs the procedure of signing data "/a/blog/article/snacks/2015/3"

first time, and is not yet authorized to do so, but the signing interpreter of the trust schema can automatically create such authorization, as we will show below.

5.2.2 *Creating Keys*

When the interpreter cannot find a signing key that corresponds to a state of the FSM (the result of step 2 in Figure 12), it transitions to a downstream state and repeats the key searching procedure. In our example, when the interpreter realizes that there are no author keys available, it will try to find out if there is any administrator key available. If not, the FSM will continue to transition downstream and repeat the search, until there are no more possible transitions available (note that self-loop transitions are skipped in the signing process when the signing key does not exist in the private key store). At this point the interpreter aborts the signing operation, as it will not be able to sign anything that will conform to the trust model.

In our example, the signing interpreter has access to the administrator's key (i.e., the author is also an administrator of the blog), and it will try to create a new author key. In order to create such key, the interpreter must derive a name for the key. In this case, the wildcard specializer `[func]` (Table 1) in a key name pattern can expand to specialize the key name. For example, `[user]` can specialize the name component for the author identifier using the local user name (e.g., "`Alex`"), and `[id]` can generate a unique identifier for the key. Therefore, at step 4 in Figure 12 (dotted blue lines), the interpreter can expand the author key pattern into "`/a/blog/author/Alex/KEY/40`". At this point, the interpreter is ready to generate an author key that satisfies the crypto requirements and overall trust model specified in the schema (step 4 on Figure 12), after which it will be ready to sign data packets of the article by this author (step 5 on Figure 12).

6. DISCUSSION

Having described the trust schema and its applications, in this section we discuss the lessons learned, ongoing efforts, and remaining research issues.

6.1 Design Pattern for Security

A trust schema is more than just an approach to describe the relationships between data and key names, it also represents a design pattern to implement NDN security. Similar to design patterns in software engineering [10], which provide general reusable solutions to commonly occurring problems in software design, the trust schema provides a reusable solution of applying commonly used trust models in NDN applications. Security experts can define a set of trust schemas as the security patterns for frequently used data authentication models. An established set of trust schemas can greatly reduce the burden on NDN application developers, who can select an appropriate security pattern for their applications during the design phase, to gain all the benefit of NDN's built-in security features.

6.2 Trust Schema Retrieval

A trust schema can be represented as NDN data packet(s), i.e., it can be named and signed. In this paper, we do not define a particular naming convention for trust schema. A meaningful name of a trust schema should be related to the name of the corresponding trust anchor, so that once a consumer learns a trust anchor, the consumer can retrieve and authenticate the trust schema.

Representing trust schema as NDN data packet allows multiple trust schemas to be combined (or chained) together: one can define a meta trust schema to authenticate other trust schemas. For example, an operating system manufacturer can use this feature to limit software installation, execution, and access to private key stores on the operating system only to applications with authenticated trust schemas. This is similar to the existing application sandboxing approaches (such as Apple's App Store and Google's Google Play), but gives operating system additional flexibility in controlling applications.

6.3 Key Caching & Bundling

In our examples, data authentication processes walked through the complete authentication paths defined by the trust schema. However, these processes can be optimized by utilizing cached keys that have been authenticated, given that a single key usually signs multiple data packets (e.g., an author uses the same key to sign multiple articles, an administrator uses his key to sign keys of authors, etc.). An interpreter can cache each intermediate key of an authentication process at the state where the key is checked and verified, so that a new authentication process may find one of its intermediate keys in those states before reaching a trust anchor.

Note that even with key caching, the first authentication process may still involve several round trips of retrieving intermediate keys. This process can be further optimized by having the data producer to maintain the chain of intermediate keys and making it available in form of a key bundle. By retrieving a key bundle, an authentication interpreter obtains all the required keys in a single retrieval. In fact, it has been a common practice in existing authentication systems (such as TLS [9]) to keep a complete chain of keys at the key owner side.

6.4 Multi-Path Authentication

Trust models that define single authentication path for data (e.g., PKI [8], DNSSEC [2]) are a common concern in the security research community. Having just one way to authenticate data creates a single point of failure, e.g., failing to timely renew certificate of any of the intermediate keys will result in data authentication failure. When multiple authentication paths are available, allowing any of the paths to authenticate data improves security resiliency of applications to maintenance failures. At the same time, by imposing a requirement that a key must be authenticated through a certain number of paths, applications can mitigate the damage of key compromise.

If/when a data packet can carry multiple signatures [19], a trust model defined with a trust schema can associate the data name with key names across different namespaces. One of our ongoing directions is exploration of a variety of conditions on trust rules, such as "any valid", "all valid", etc.

6.5 Trust Bootstrapping

In describing the blog website example, we assume that data consumers have already obtained the trust anchor of the website. In general, a consumer may not always be able to obtain the trust anchor for each website it visits a priori. In today's practice, a consumer may need to bootstrap trust from a limited number of pre-configured trusted keys and eventually establish trust on a particular website's trust anchor. We believe that bootstrapping trust remains as an important and challenging open issue, which is beyond the scope of this paper but included in our ongoing efforts. Besides the use of the existing Internet style PKI in NDN networks, more exciting directions to explore this open issue include realizations of web-of-trust and evidentiality-based trust bootstrapping models.

6.6 Signature Revocation

Signature revocation is an open research problem in NDN trust management. Although this problem is beyond the scope of this paper, our ongoing efforts explore the following approaches:

- constraining validity period of issued signatures, which may require mechanisms to certify validity of the signature at the time of creation (e.g., using secure timestamp);
- using trusted services to certify current validity of the signature, similar to revocation lists and OCSP in current PKI.

6.7 Formal Trust Schema Syntax

The syntax we used to describe the trust schema is still at an experimental stage. Trust schemas share many design philosophies with logic programming languages (such as Prolog [7]). It may be helpful to unify the trust schema syntax with formal syntax used by existing languages, and we would like to encourage researchers to apply techniques of programming language to enhance the trust schema design and improve the security of NDN applications.

7. RELATED WORK

The focus of this paper is trust management automation. We are aware of similar efforts for Public Key Infrastructure (PKI), including a standardized path validation algorithm for X.509 certificate authentication [8], certificate chain discovery methods for SPKI certificate system [6, 14], and general chain discovery mechanisms [3]. However, these studies assume a specific trust model. Automation based on trust schemas is a general trust management solution for NDN applications which can have different trust models. Moreover, it not only allows automation of authentication process, but also enables (at least partial) automation of the data signing process.

The design of trust schema leverages NDN naming to enforce name-based trust policies for data packets. DNSSEC [2], a security extension of DNS, adopts a similar mechanism to authenticate DNS resource records: a key bound to a DNS domain name is globally trusted to sign only DNS resource records under this domain. DANE [11] extends the name-based mechanism of DNSSEC to authenticate a TLS public keys. At the same time, both DNSSEC and DANE assume a specific hierarchical trust model, while our trust schema can capture many different trust models that NDN applications may need.

The trust schema is basically a policy language, where rules define policies on which keys are trusted to authenticate data. Compared to previous work on policy languages for access control and authorization, such as PolicyMaker [5], SD3 [12], RT [13], and Cassandra [4], our work focuses on data authentication and integrates data authentication into the NDN network architecture.

8. CONCLUSION

Usability is a fundamental requirement for any security solution. The NDN design mandates that each network-layer data packet carries a cryptographic signature for authentication. Although this requirement on the packet format represents a significant first step toward securing networked systems, its actual effectiveness depends on the implementation. Our observations during the first few years of NDN application development suggest that it is a non-trivial task for application developers to properly define trust relationships between data and keys, to handle proper key chain creation, and to enforce authentication of data according to the defined rules. It happens too often that developers use shortcuts to get around security (e.g., using hard-coded keys, or simply turning verification off "temporarily" when it blocks development progress).

In response to the above important and urgent issue, we invented the idea of a trust schema to formally define application trust models, and to automate the signing and verification processes. We developed prototypes of two trust schema interpreters that can convert trust schemas into finite state machines and help applications to rigorously sign and authenticate data automatically. We applied our prototypes to secure a range of NDN applications, and our experience so far gives us confidence in the solution's general applicability to most, if not all, NDN applications.

We believe we have contributed a meaningful step toward a reusable approach to data authentication. We plan to apply the schematized trust management in more NDN applications and integrate the schematized trust management with operating system support. We would also like to see interested parties, especially people in security research community, to identify and define other commonly reusable trust schemas ("security design patterns") for popular network applications, to be used to secure more applications across the Internet.

9. ACKNOWLEDGEMENTS

This work was supported in part by the NSF grants CNS-1345318 and CNS-1345286.

10. REFERENCES

[1] A. Afanasyev, C. Yi, L. Wang, B. Zhang, and L. Zhang. SNAMP: Secure namespace mapping to scale NDN forwarding. In *Proc. of Global Internet Symposium*, 2015.

[2] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS security introduction and requirements. RFC 4033, 2005.

[3] L. Bauer, S. Garriss, and M. K. Reiter. Efficient proving for practical distributed access-control systems. In *ESORICS*, 2007.

[4] M. Y. Becker and P. Sewell. Cassandra: Distributed access control policies with tunable expressiveness. In *Proc. of International Workshop on Policies for Distributed Systems and Networks (POLICY)*, 2004.

[5] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proc. of IEEE Symposium on Security and Privacy*, 1996.

[6] D. Clarke, J.-E. Elien, C. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, 2001.

[7] W. Clocksin and C. S. Mellish. *Programming in PROLOG*. Springer Science & Business Media, 2003.

[8] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. RFC 5280, 2008.

[9] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.2. RFC 5246, 2008.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[11] P. Hoffman and J. Schlyter. The DNS-based authentication of named entities (DANE) transport layer security (TLS) protocol: TLSA. RFC 6698, 2012.

[12] T. Jim. SD3: A trust management system with certified evaluation. In *Proc. of IEEE Symposium on Security and Privacy*, 2001.

[13] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust-management framework. In *Proc. of IEEE Symposium on Security and Privacy*, 2002.

[14] N. Li, W. H. Winsborough, and J. C. Mitchell. Distributed credential chain discovery in trust management. In *Proc. of Conf. on Comp. and Comm. Security (CCS-8)*, 2001.

[15] NDN Team. Libraries / NDN platform. `http://named-data.net/codebase/platform/`, 2015.

[16] NDN Team. NDN packet format specification. http://named-data.net/doc/ndn-tlv/, 2015.

[17] NDN Team. NDN regular expression. `http://named-data.net/doc/ndn-cxx/current/tutorials/utils-ndn-regex.html`, 2015.

[18] D. Smetters and V. Jacobson. Securing network content. Technical report, PARC, 2009.

[19] Y. Yu. Public key management in Named Data Networking. Tech. Rep. NDN-0029, NDN, 2015.

[20] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, kc claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang. Named data networking. *ACM Computer Communication Reviews*, 2014.