Trust in Information-Centric Networking: From Theory to Practice

Christian Tschudin
Dept of Mathematics and Computer Science
University of Basel, Switzerland
christian.tschudin@unibas.ch

Ersin Uzun
Palo Alto Research Center
Palo Alto, CA 94304, USA
ersin.uzun@parc.com

Christopher A. Wood[+]
Dept of Computer Science
University of California Irvine, USA
woodc1@uci.edu

***Abstract*—We present the logical design of a trust engine for Information-Centric Networking (ICN) that is capable of efficiently and correctly verifying content integrity and authenticity. Our primary contribution is the synthesis and unified treatment of four different and popular trust models. We show in which operational aspects they vary and emphasize which parts of the verification mechanics are invariant. The verifier logic is expressed in Prolog to show its simplicity (abstracting away, e.g., procedural certification chain verification steps) and to highlight subtle errors that can occur in the use and enforcement of trust models. The details of an implementation of our trust engine in the CCNx network stack are presented to demonstrate its viability and general modularity. A simplistic interface enables the trust engine to be easily ported to any ICN-style network software. Finally, we demonstrate how application instantiations of various trust models are natively supported by the trust engine to illustrate its flexibility.**

***Index Terms*—Information-centric networking, Trust models, Trust engine, Content object verification**

I. Introduction

Today's Internet architecture closely resembles legacy packet-switching communication networks. While this design was once well suited for communication-related applications such as e-mail and access to remote computing resources, the demands of modern mobile and data-centric applications from the network are significantly different. In response to these problems, a number of independent research efforts have spawned to design the future Internet architecture. Content-Centric Networking (CCN) [4] by PARC and its academic fork Named-Data Networking (NDN) [15] are two examples that employ an information-centric approach to networking (ICN)[1].

In contrast to today's IP-based networks, CCN names content rather than hosts in the network. In other words, CCN transfers named-data (content) instead of packets addressed to particular hosts. Every network level data object is individually addressable and is signed directly or indirectly by a digital signature. This radical change in the naming and security model enables content to be opportunistically cached and served from any node in the network while still providing the following guarantees at the packet level:

- *Correctness*: Signatures securely bind content names to content payloads enabling secure verification of received content with a name that matches the request.
- *Integrity*: A valid signature ensures the content, as received, was not modified in transit.
- *Authenticity*: Signatures are tied to content producers via their public keys. Thus, a valid signature identifies the origin (producer) of the content.

In practice, however, this verification places a non-negligible burden on networks nodes. For instance, signature verification is computationally expensive. So even if routers had the correct verification keys, they cannot be expected to verify the signatures of all forwarded content without degrading the throughput. Moreover, trust is an application-specific concept and intermediate nodes in the network usually lack the contextual information needed to determine which content should be trusted.

Opting out of the security verification process in CCN is not practical as it leaves the network susceptible to cache pollution and denial of service (DoS) attacks [7], [9]. Ghali et al. discuss this issue in detail in [9] and identify the minimum functionality needed at the network layer to mitigate such attacks. The two main conclusions they arrive at are: (1) requests in CCN should carry sufficient information about the consumer's trust context and (2) CCN nodes should assert whether or not a content aligns with the requester's trust preferences before using it as a viable response to a request.

Given the critical importance of trust management and enforcement, the above suggestions are already employed in the latest protocol specification of CCN [16] to provide the basic functionality for network-layer trust enforcement. However, some key issues have yet to be resolved regarding *practical* trust management for routers and end hosts in CCN:

- Security guarantees are provided at the network level in CCN. However, there is no standard model or pattern for implementing and enforcing *application-specific trust models* at end hosts in CCN.
- There is no concrete machinery via which complex *application-specific trust semantics* can be tied to the basic enforcement functionality provided by the network.

In this paper we address these issues by first capturing the requirements of and describing the logical design for a trust engine that is capable of efficiently verifying content authenticity under different trust models. This trust engine extends network verification with the required functionality to implement and enforce application-specific trust models. Second, we present one possible implementation of this engine

[+]Supported by NSF Graduate Research Fellowship DGE-1321846.

[1]In this paper, we focus on CCN for a more coherent presentation due to minor differences between the two sibling architectures.

as a user-defined and configurable trust management module for the CCNx [4] network stack. We then show how to specify and enforce several popular trust models using this engine, including the recently schematized trust models pioneered by NDN [19], [20]. To the best of our knowledge, this is the first attempt to detail a high-level guiding logical design and an actual implementation of it within a network stack.

II. CCN Architecture Overview

The fundamental principle of data transfer in CCN and related architectures is simple: *named content objects* are transferred from producers to consumers in response to *explicit requests*. Such requests are formulated as interest messages that specify the intended content object's name. For example, an interest for an image `fileA` published by entity `entityX` could have the name `/entityX/images/fileA`.

Interests have optional fields called `KeyId` and `ContentObjectHashRestriction`. The `KeyId` field is used to specify the (cryptographic) hash digest of the public key that is trusted by the requester to verify the signature of the requested content object. Similarly, the `ContentObjectHashRestriction` field is used to specify the hash digest of the desired content object. Together, these fields limit the set of possible objects that can be returned in response to an interest message and allow consumer trust preferences to be enforced at the network layer [9].

Structurally, content object messages have the two following important differences from interest messages: (1) content objects must carry a payload, and (2) they *do not* have a `ContentObjectHashRestriction` field (this hash is calculated at each hop). In addition, they also carry an optional validation payload which holds the content signature or some alternative form of authenticator, e.g., a HMAC tag.

To enable the transfer of named data, CCN routers have forwarders which are composed of (1) a forwarding information base (FIB) and (2) pending interest table (PIT). Forwarders may also have an optional content store (CS). Much like traditional IP routing tables, the FIB is populated using standard routing protocols or static routes and matches interest names to FIB entries using longest prefix match (LPM). The PIT stores information of previously forwarded interests so that content object responses may follow the reverse path back to the requester. (This is why interests do not carry a source address.) This information includes the interest arrival interface(s), name, `KeyId`, and `ContentObjectHashRestriction`. Finally, the CS is a cache for content objects that, if present, is first searched for matching content objects prior to forwarding an interest upstream. If the CS contains the matching content object for an interest, then the interest is consumed and the match is forwarded downstream.

A CCN forwarder will only satisfy an interest via a content object from its CS or from upstream if the names (and the `KeyId` fields if specified in the interest) are equal to that of the content object. Additionally, if the `ContentObjectHashRestriction` is specified in the interest packet, a CCN forwarder will also calculate the hash of the content object and check for equality with the value provided. CCN forwarders with a CS are not allowed to respond to interests that specify a `KeyId` from their CS unless they mechanically verified the signature and the `KeyId` on that content object are correct.

As a final note, CCNx 1.0 includes Manifests, a type of content object that encapsulates pointers to other content object chunks and metadata for these pointers (see [3], [13], [18]). Each pointer contains an optional name and a `ContentObjectHashRestriction` value that is used to issue an interest for the corresponding chunk. This avoids the need to verify the signature on every chunk. That is, upon the validation of the digital signature on the manifest, the list of pointers (with names and hashes) it carries can be used to issue interests with the specified `ContentObjectHashRestriction` values so that the responses can be verified by hash equality.

III. The Importance of Trust in CCN

CCN allows content to be cached and served from any forwarder in the network. Without proper trust enforcement at the network layer, fake or corrupted content objects (i.e., content poisoning attacks) would have a grave effect on the network [7], [8].

Ghali et al. [9] correctly observed that network layer trust in content and content poisoning are inseparable in CCN. Consequently, routers must be given some way to ascertain the authenticity of content objects in response to interests. Thus we abide the rule defined in [9] which states that each interest must reflect the trust context of the issuing consumer in a form enforceable at the network layer. In CCNx 1.0 this translates into the need for an interest message to uniquely identify the content object signature verification key (via the `KeyId`) or the hash digest of the requested content object (via the `ContentObjectHashRestriction`). This rule identifies an architectural design element that is necessary for CCN to be robust against content poisoning attacks. However, it is far from sufficient to achieve the robustness, security, and flexibility that is required for applications. To date, there is no clear mapping from application trust models to application-agnostic enforcement mechanisms.

When translating the theory of trust in CCN to practice, we claim that one must consider at least the following:

1) Given the complexity of the trust verification process under potentially many indirections and trust models, a complete logical design for the process is necessary to guide successful implementation efforts.
2) Since trust is an integral part of the network architecture in CCN, enforcement logic and mechanisms should not have to be encoded at the application layer.

Thus, a *trust engine* for the network stack and corresponding APIs are needed to make security a seamless part of development, independent of what trust model an application might choose to employ. In this paper, we aim to bridge this the gap from theory to practice by designing the overall guiding logic to implement trust management in CCN-style networks and describe our implementation of it in the network stack. We also provide examples for initiating various trust models using our design and implementation.

IV. Trust Management (Four Standard Trust Models)

The core elements of any trust management scheme are, at a minimum, (a) specifying and enforcing trust models and (b) bootstrapping and maintaining trust context for this enforcement. In this work, we consider a trust model to be a set of rules by which content is both *signed and verified*. Thus, content signed under one trust model must be verifiable in the same trust model. Bootstrapping is the process of initializing the state of the trust context, which is a set of trusted keys and procedures that can be used to procure more trusted keys. In this way, the trust context organically grows in time.

Logically, trust models can either be centralized or decentralized (the implementations will typically be distributed, though). Centralized trust models are similar to PKI schemes where certifying authorities bind subordinate keys to identities via digital signatures. This creates a chain of keys that are rooted at *trust anchors*. In a **hierarchical trust model**, certificates and signing authorities follow a hierarchy that is rooted at one or more central anchors trusted by all verifiers. Practical instantiations of this trust model might rely on x509 certificates to explicitly identify the names of parent keys and to restrict the name space for which a signed key is valid.

The **schematized trust model** introduced in [20] is a restriction on the hierarchical trust model where key certificates and other signed content cannot have arbitrary names but must obey some well-defined pattern that links the name of the signed key or the content to the name of the signing key. As in the hierarchical case, the central element is the trust anchor, plus the set of rules specifying the acceptable linkage between the names for that schematized trust model instance. Schematized trust models can be useful for intra-organization trust management where the ownership of the entire schema namespace is not an issue.

Contrary to centralized trust models, distributed trust models do not require keys to be generated by centralized parties. The **Web-of-Trust model** is one example of a distributed trust model wherein keys are still connected by some chain, but the trust of each link in this chain is not a binary decision (i.e., is the signature from the certifying authority valid?). Instead, keys are associated with measures of confidence. (More details about computing this metric can be found in PGP-like recommendations [1], [17] and algebraic trust quantization outlined in [11]). The implication of these models is that verifiers must compute the measure of trust in keys. If the computed confidence in a key falls below a given threshold, then the key is considered untrusted and verification fails. Such logically distributed models are appropriate for applications where there is insufficient confidence in the existing PKI environment and the certifying authorities contained therein.

Finally, consider the case where the validity of content is not ascertained using public key digital signatures, but through a pre-shared symmetric key. In CCN, the validation payload of a content object may contain a keyed Message Authentication Code (MAC), e.g., HMAC-SHA256. To authenticate the content object, a validator must possess the symmetric key used to generate the MAC tag; there is no external key resolution as with the trust models based on public key cryptography. Functionally, keyed MACs provide a different level of trust guarantee to verifiers than the digital signatures due to its symmetric nature (i.e., the *same* key is used by both the generator and the verifier). HMAC authenticators are appropriate in scenarios where the task of performing public key signing or verification is too expensive (in terms of computation, memory, or even power). Thus, one plausible ecosystem wherein pre-shared key trust models might be used are sensor networks with a fixed and limited set of devices, all of which lack the computation and power requirements for public key cryptographic operations.

Note that MACs are *not* cryptographic signatures. However, for presentation purposes, we will refer to HMAC tags as signatures and to tag generators as signers since, for the purposes of verification considered here, both HMAC tags and signatures are used to decide whether the received content is intact (i.e, not modified in transit) and authentic.

In the following section we will refer to these four prototypical cases as the **keyed MAC**, **hierarchical**, **schematized**, and **Web-of-Trust** trust models.

V. Trust Model Templates for a Trust Engine

In this section we specify the logic of a general trust engine that can handle the four trust models introduced above. As we will show later, the logical details for each model are not that different despite the (perceived large) differences in practice. We will use Prolog to express the validation and signing logic and point out how this high-level representation can also be used to search for conceptual and configuration problems in a given trust model instance.

A. Core Validation Logic

As previously stated, a trust context in which a packet must be validated is merely a list of trusted keys plus a model-specific (set of) procedure(s) for procuring additional trusted keys on demand. Signature validation is then carried out by first trying to locate a suitable key in the trust context and only if this fails by fetching additional keys in a controlled way. *Bootstrapping* the trust context means starting with a minimally configured trust context containing only a few trust anchors that then is inflated at run time. The trust context also incorporates the concept of *caching keys*: The expensive fetching of keys can be amortized by recording the outcome of each key validation decision.

We now turn our attention to expressing the core validation logic in Prolog. There are several benefits to this representation. First, Prolog permits us to hide iteration: Looping over a list of trusted keys, for example, does not need to be coded explicitly. Second, as we will see for schematized trust, rules that operate on name (positional) patterns are quite easy to express without referring to `grep` or other tools working at the text surface. As a reminder, in Prolog, variables start with an uppercase letter while the underscore character is a placeholder for any value. Now, finally, using Prolog, the trust engine's core logic for packet validation is surprisingly simple and can be expressed as in Figure 1.

```
isValidPkt(Packet, TrustContextIn, TrustContextOut) :-
  Packet = pkt(DataName, _, KeyInfo, PktHash, PktSignature),
  getTrustedKey(DataName,KeyInfo,TrustContextIn,TrustContextOut),
  KeyInfo = key(_, _, KeyBits),
  isValidSignature(PktHash, PktSignature, KeyBits),
```

Fig. 1: `isValidPkt` predicate.

Starting with a given packet and trust context, the system tries to satisfy the `isValidPkt()` predicate by getting a trusted key and validating the packet's signature.

As can be seen in the code sample above, some data structures are made explicit: A *packet* has a field for the data name, the information regarding which key was used to sign, the hash value (computed over the received bytes) and the signature value added by the packet producer.

We use three fields to represent *KeyInfo*; namely the key's name, the key's ID, and the key's value (raw bytes). Note the use of the underscore to leave some fields empty: A packet could include a key locator, key ID, or nothing and yet still have the KeyInfo.

Finally, the *trust context* contains a tag telling which kind of trust model it represents, the list of trusted keys, plus an auxiliary field for the name pattern rules (schematized trust) or the list of acquainted names of peers and their trust level (web of trust).

Getting a trusted key may include the side effect of inflating the trust context; this is the step wherein different trust models lead to different run time behavior. The `getTrustedKey` predicate must handle two cases: (1) either we find a suitable key in our context (using the `member()` function), in which case the trust context does not change (i.e., the input and output parameters are identical), or (2) we have to fetch a suitable key over the network in a trusted way. This is expressed in a (still) model-agnostic way with the following two rules:

```
getTrustedKey(_, KeyInfo, TrustContext, TrustContext) :-
  TrustContext = trustCtx(_, TrustedKeyList, _),
  member(KeyInfo, TrustedKeyList).

getTrustedKey(DataName,KeyInfo,TrustContextIn,TrustContextOut) :-
  fetchTrustedKey(DataName,KeyInfo,TrustContextIn,TrustContextOut).
```

Fig. 2: `getTrustedKey` predicate rules.

All differences among the trust models are now isolated to the `getTrustedKey()` predicate, for which we now show the different incarnations.

B. Validating Keyed MAC Signatures (Tags)

In the simplest trust model, HMAC, keys are pre-shared among the producer and validators and there is no way to enlarge the set of trusted keys at run-time without invoking an on-demand key exchange protocol [14]. Consequently, the `fetchTrustedKey()` in Figure 3 is an always failing action.

```
fetchTrustedKey(_, _, Context, Context) :-
  Context = trustCtx('preshared', _, _), fail.
```

Fig. 3: `fetchTrustedKey` predicate for the HMAC model.

An interesting point is how the KeyID value, typically present in HMAC-signed packets, is used. From the `isValidPkt()` code it can be seen that this value is never consulted (it is the second field of a `key(Locator, KeyID, KeyBits)` data structure). Prolog will simply try out all trusted keys until validation succeeds. This means that the KeyId is just an optimization helping a concrete implementation to narrow down the number of keys to try.

C. Validating Signatures in the Hierarchical and Schematized Trust Models

Recall that schematized trust is basically a hierarchical key certification system with the addition that the name of the signed key and the name of the signer have to obey some constraints (i.e., name schemas). This is manifested at the code level where the core logic of these two trust models basically differ by one Prolog line expressing this constraint on names. The code sample in Figure 4 shows the corresponding `fetchTrustedKey()` predicate that is used for both trust models.

```
1  fetchTrustedKey(DataName,KeyHint,TrustContextIn,TrustContextOut) :-
2    KeyHint = key(KeyLocator, _, KeyBits),
3    TrustContextIn = trustCtx(Model, _, Aux),
4    ( Model = 'hierarchical'
5    ;
6      Model =  'schematized', % Aux has the list of schemas
7      member(schema(KeyLocator, DataName), Aux)
8    ),
9    ccnFetchCert(KeyLocator, CertPkt),
10   CertPkt = pkt(KeyLocator, KeyBits, _, _, _),
11   isValidPkt(CertPkt, TrustContextIn, TrustContextTmp),
12   TrustContextTmp = trustCtx(Model, KeyList, Aux),
13   TrustContextOut = trustCtx(Model, [KeyHint | KeyList], Aux).
```

Fig. 4: `fetchTrustedKey` predicate for the hierarchical and schematized models. Lines 4-7 refer to the differences in the two models where the name constraint is applied.

Note the reference to `ccnFetchCert()`: An actual implementation will issue an interest packet for the given KeyLocator and receive back a content object whose data is the corresponding key bits. Because this packet is signed, the packet itself is a certificate for that key, which we recursively validate by calling our main `isValidPkt()` predicate. Furthermore, in the real implementation, the certificate that is returned will be checked for expiration and possibly revocation, among other properties. In this way, a certificate chain is established between the original packet to validate our trust anchor.

D. Validating Signatures in the Web of Trust Model

Validating a packet in the web-of-trust model resembles (code-wise) the hierarchical and schematized models because a certificate chain is established here as well. However, web-of-trust is (search-wise) closer to the keyed MAC model in the sense that we have to blindly search for valid signed keys (a key ID does not help us here as it points to the signer, while we have to explore in the other direction): Starting from our list of peers whom we trust we search for their peers and their keys (as shown in Figure 5).

The `fetchTrustedKey()` predicate for the web-of-trust model is shown in Figure 5. In this example, we use a simple

```
fetchTrustedKey(_, Key, TrustContextIn, TrustContextOut) :-
  TrustContextIn = trustCtx('webOfTrust', KeyList, ConfidenceList),
  member(confid(FriendName, Conf), ConfidenceList),
  (
    Conf < 0.5, fail
  ;
    ccnFetchFriends(FriendName, FriendList),
    member(FriendFriend, FriendList), % for all peer's peers ..
    not(member(key(FriendFriend,_,_), KeyList)), % if new, do:
    ccnFetchCert(FriendFriend, CertPkt),
    CertPkt = pkt(FriendFriend, KeyBits, _, _, _),
    isValidPkt(CertPkt, TrustContextIn, TrustContextTmp),
    Key = key(FriendFriend, _, KeyBits),
    TrustContextTmp = trustCtx('webOfTrust', KeyList2, ConfList),
    C is Conf * 0.9,
    TrustContextOut = trustCtx('webOfTrust', [Key | KeyList2],
                               [confid(FriendFriend, C) | ConfList]),
    !    % one key suffices (social graph may have loops)
  ).
```

Fig. 5: `fetchTrustedKey` predicate for the web-of-trust model.

trust computation which factors in a confidence value that decreases the more remote a signer is in our social graph. For this trust model, the trust context contains (in its third position) a list of tuple with a signer's name and its measure of confidence.

If during packet or certificate validation no suitable key is found in our trust context, we will use all signer names in the confidence list that are trustworthy enough and ask for their peers' names through `ccnFetchFriends()`. For each new name we fetch the corresponding certificate, validate it, and add it to our list of trusted keys. In doing so, we add that new peer and its confidence value to the ConfidenceList.

E. Signing

Signing *data* packets usually does not warrant much discussion since a running program (producer) typically knows its identity and key. However, a producer must be careful about the packet it signs, as a signed content object in CCN can become a certificate. If a producer were to sign packets with arbitrary names, some third party could get a certified *key* that inherits all trust properties of the producer (in a hierarchical trust model). One solution is to opt for x509 certificates which state for which prefix a key can be used. Another path is to use schematized trust and to impose restrictions on the name relation between signed key and signer's key.

A second observation is that a producer can have more than one key signed by the same signer (e.g., during a key rotation), or multiple keys from different signers and trust models, as was pointed out in [20]. Both cases – checking packet names before signing and finding keys that will permit validators to verify the signature for some given trust model – are covered by the code excerpt in Figure 6.

```
getSigningName(NameToBeSigned, TrustContext, [SignerName|Tail]) :-
  TrustContext = trustCtx('schematized', KeyList, Schema),
  member(schema(SignerName, NameToBeSigned), Schema),
  (member(key(SignerName, _, _), KeyList), Tail= []
  ;
   getSigningName(SignerName, TrustContext, Tail)
  ).
```

Fig. 6: `getSigningName` predicate.

The important lines are the ones doing the `member()` checks, first for finding a suitable schema rule and then for picking a valid certification path. As a convenience, the `getSigningName()` predicate returns that certification path for the selected key name. Note that for readability of the high-level code we do not distinguish public from private key bits in our KeyList data structure.

F. Using Prolog Code to Check Trust Model Properties

A danger with certificate chains is that the validator has to follow a sequence of "pointers" which cannot be fully trusted. For example, by negligence, complexity, or malicious action, loops could arise in a chain. Even with schematized trust, unsafe configurations can be created, as we show in Figure 7. (The appendix provides the code that can be used to demonstrate this problem case.)

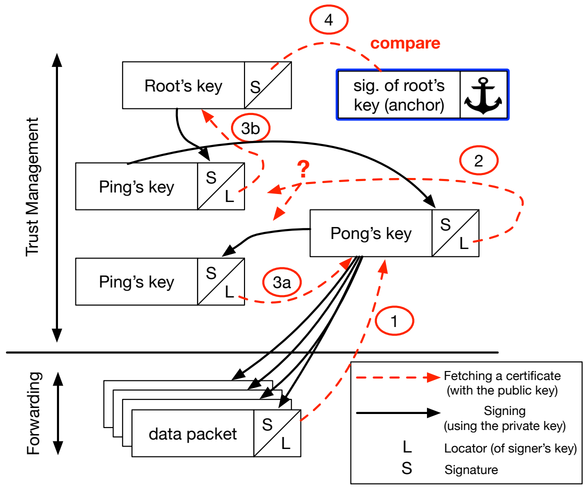
4
compare
Root's key
S
sig. of root's key (anchor)
3b
Trust Management
?
2
Ping's key
S
L
Pong's key
S
L
Ping's key
S
L
3a
1
Forwarding
data packet
S
L
Fetching a certificate (with the public key)
Signing (using the private key)
L Locator (of signer's key)
S Signature

Fig. 7: A chain of (valid) schematized certificates with a loop that arises when Ping's key is signed by both Pong's key and the Root's key.

In Figure 7 there are three principals called "Root," "Ping," and "Pong." We assume that the schema would state that data packets have to be signed by Pong and that Pong's key must be signed by Ping. Finally, Ping's key has to be signed by Root and can, optionally (justified by test and redundancy concerns) also be signed by Pong. The black arrows depict a signing action while the (dotted) red "upwards" arrows show the certification chain towards the trust anchor.

Initially, the trust context (upper part of Figure 7) only contains the trust anchor e.g., in form of a message digest of Root's key. Upon validation of a first data packet signed by Pong, the locator found in that packet is used to retrieve the signer's key (step 1). Having retrieved the certificate (and public key) of Pong's key, we find another locator, this time for Ping's public key. This certificate is fetched in step 2, followed by a fetch of the Root's public key in step 3b. Having retrieved Root's public key, the validator can compare its digest with the trust anchor value and complete the verification of the trust chain. All subsequent packets can be authenticated without having to fetch any keys if the validator caches the acquired trust in Pong's public key.

A loop can arises if Pong signs Ping's key (which is used to sign Pong's key, and therefore form the loop)[2]. This means that Ping's key has two certificates: When following the certificate chain, it could happen that the `ccnFetchCert()` systematically returns Pong's certificate earlier than the certificate issued by Root (step 3a). This will disrupt the validation process since we cannot follow the trust chain up to the trust anchor. It is important to observe that all certificates are valid and no principle cheated. The loop is a pure configuration error which, for resource protection reasons must be handled as needed.

In practice, schemas either have to exclude the possibility that two authorities can (indirectly) sign the same key, or the schemas have to be scrutinized before putting them in place. Detecting loops in schema rules is not difficult but complex enough that this should be checked by programs, not by humans, which is where schemas expressed in Prolog can help.

G. Using Prolog to Demonstrate a Problem with Weak Certification Pointers

One obvious problem in certification chains is that fake data can be created if trust is subverted. For example, an attacker can try to insert fake or bogus certificates. Referring to Figure 7 again, assume that the root of trust starts to misbehave. A simple attack, which we call *certification poisoning*, is that the root principle invents a key for Ping, using random bits, and then issues a valid certificate for it. The effect is that when the packet validator traverses the certification chain and wants to check Pong's key, two keys can be returned by the network: The correct one and the bogus one, both having a valid signature by the trusted Root. If Root now manages to inject the bogus key (by putting it on a faster upstream node than the place which has the correct key), this key will be pulled and cached downwards along the certification chain. Since the key consists of random bits, neither Pong's key or any packets signed by Pong can be validated.

This attack exploits the fact that names are not (cryptographic) identifiers. One must assume that certificates are fetched by only the name for the validator to acquire an incorrect key (i.e., one whose name is retrieved from the signature of Pong's key). If, however, we add some self-certifying element to the key's name, for example using the key's digest as the `ContentObjectHashRestriction` value, this attack can be prevented. Having retrieved Pong's certificate we find an unambiguous key identifier which consists of a name that is relevant for routing and the key's hash that is responsible for selecting exactly the key that was used to sign Pong's key instead of the bogus key produced by Root.

We have verified the existence of both problems above loop case and certificate poisoning attack in Prolog and compared it to actual implementations. NDN [15], for example, defines the KeyLocator to be either a name or a digest, but not both at the same time:

```
KeyLocator ::= KEY-LOCATOR-TYPE TLV-LENGTH (Name | KeyDigest)
```

Our recommendation therefore is that in NDN a signing key's name MUST include the key digest in some form to foil the certificate poisoning attack[3].

VI. A Trust Engine Implementation

Trust management and enforcement should be, to some extent, transparent to the application developer. Given a trust model, its enforcement should happen more or less automatically beneath the application. In this section we present the design of the CCNx trust engine which is capable of such autonomous behavior based on the previous logical design. We claim that the engine is sufficiently general to be used elsewhere, e.g., as a co-processor in a forwarder used to filter invalid or untrusted content before storing it in a content store. Where appropriate we also refer to internal data structures (e.g., whitelists) of our implementation to provide insights into a practical trust engine.

A. Packet Processing Pipeline and Trust-Specific FSM

As illustrated in the previous sections, verification of a single packet follows a simple and general strategy, regardless of the trust model required. As exemplified in [20], this strategy can be implemented as a finite state machine (FSM), wherein there are three simple states:

1) InspectPacket: Extract the necessary information from the input packet pairs (i.e., an interest and corresponding content object).
2) FetchTrustedKey: Obtain the key required to verify the packet data extracted from the InspectPacket state. The logic of this state varies depending on the type of trust model required. Furthermore, this state may mutate the current *trust context* for all future packet verification attempts.
3) VerifySignature: Verify the signature of the input packet from the InspectPacket state using the trusted key obtained from the FetchTrustedKey state.

Packets are processed in pairs since the validity of a content object depends on information provided in the corresponding interest. As shown in Figure 8, these pairs can come from (a) an input queue (IPQ) connected, e.g., to a lower packet processing component such as the forwarder, or (b) a pending packet queue (PPQ) (to be explained later). When the PPQ is empty, and there exists a packet in the IPQ, the trust engine will dequeue the first element in this queue for processing. Before running this packet through the verification FSM described above, it will first be checked against a whitelist and blacklist of trust sources. Each entry in the whitelist is a name (or prefix) and flag ACCEPT or ALLOW, whereas each entry in the blacklist is just a name (or prefix). If the packet exists in the whitelist with the flag ACCEPT, the packet is accepted

[2]Good security practice separates data-signing-keys (DSK) from key-signing-keys (KSK), which is not done here. However, this distinction could easily be included in the already complex figure by inserting a DSK signed by Pong, but would not change the loop case discussed here.

[3]In contrast, the CCNx specification (https://tools.ietf.org/html/draft-irtf-icnrg-ccnxmessages-02) states that *"These Validators require a KeyId* **and** *a mechanism for locating the publishers public key (a KeyLocator)"*. In this quote (where we added the emphasis), KeyLocator refers to the name of the key, not the full data structure as in NDN.

(assumed to be correct) without verification. If, however, this flag is ALLOW, then the packet will be subjected to further processing by the verification FSM. Conversely, if the packet is not in the whitelist but in the blacklist, then the packet is rejected automatically without further processing.

Once a packet reaches the verification FSM, it runs through each of the aforementioned states as expected. During the FetchTrustedKey step, it is possible that the key that must be used for verifying the input packet cannot be obtained locally, i.e., from the local key store. This occurs if the key has not yet been requested and is linked from the input packet by a KeyLocator. In this case, the engine will do the following:

1) Request the desired key using the locator provided in the content object.
2) Defer the current packet pair under inspection to the Deferred Packet Queue (DPQ) with the desired key name.
3) Start processing the next packet in the IPQ or PPQ.

The purpose of the DPQ is as follows: When the key needed to verify a content object cannot be located it must be fetched from the network. Since this request returns yet another content object which must be verified, the trust engine must be able to process packets recursively. Only upon verifying the key content object response can the trust engine resume processing the original packet. This recursive behavior is supported with the DPQ: when a packet is verified, resulting in acceptance or rejection, the DPQ is scanned for entries that depend on the outcome of said packet. If there exists such a packet, they are moved to the PPQ (in the case of acceptance) or dropped (in the case of rejection).

A trust engine that implements this behavior is shown in Figure 8. In practice, the actual implementation requires only the following four functions to operate correctly. These are labeled in Figure 8.

1) `process`: This is called to insert a new packet pair into the trust engine to be processed.
2) `accept`: This is called when a content object has been verified and is deemed trustworthy according to the specific trust model.
3) `reject`: This is called when a content object fails verification.
4) `request`: This is called when additional key(s) need to be acquired to finish verifying some content object.

B. Runtime Modifications

The engine has a simple command-based interface that allows command messages to be asynchronously sent from clients to control its behavior at runtime. A command is an object with an identifier (string) and payload (data). For example, to add a trusted root public key with no associated schema to the current trust context of the engine (e.g., the key store), a JSON-encoded command with the following representation is sent to the engine:

```
{CMD : "ADD_ROOT", PAYLOAD : {
    digest : <public_key_digest>,
    local : false,
    schema : null,
}}
```

The trust engine API implementation parses the command message, extracts the identifier and associated options payload, and performs the required operation (i.e., adding a trust root). The trust engine API supports a variety of commands to which it must respond or handle at runtime. A subset of these are detailed below.

- Whitelist and blacklist modification: This family of commands serves to (add or remove) (whitelist or blacklist) sources from the basic verification checks. In particular, the payload for one of these commands contains a set of keys, each of which map to a set of namespace prefixes that will be (added to or removed from) the (whitelist or blacklist) source. Variations of these commands are available to support name schemas instead of just prefixes.
- Set the trust engine rejection action: The default behavior for the trust engine is to automatically discard messages upon verification rejection since they cannot be authenticated. Application developers may wish to change this behavior to, for example, permit such messages to be considered trusted so as to subject them to further processing above the trust engine. This command allows them to do so.
- Toggle FetchTrustedKey recursion depth n: This command serves to set a limit on the number of recursive calls a given verification step can take in the trust engine. This can be used, for example, to enforce that the trust engine may only attempt to fetch at most n keys to verify a given content object before giving up and rejecting the packet.
- Add schema rule to the trust engine context: This command specifies a new signing key name and content object name relationship to be added to the set of rules in the trust context. The syntax for schema rules follows that which is specified in [20].

The simplicity of the trust engine API (e.g., a single function call that accepts JSON-encoded commands) is both flexible and extensible. It enables new commands for behavioral modification to be easily added as needed. This permits application developers to experiment with more sophisticated enforcement policies in the trust engine.

VII. Trust Model Instantiation in Practice

The core verification logic requires trust context from an application in order to perform automatic verification (on consumers) and signing (on producers). To specify a model (or mixture of models), the trust context must convey the parameters for the key fetch routines as described in the previous section. The trust engine will execute the verification and signing logic using the trust context to perform automatic verification and signing. Thus, all that is required from an application are (a) the set of trusted keys, and (b) the trust model and associated key fetch parameters. To illustrate the ease by which the trust engine can be used, we show how four general applications may perform tasks (a) and (b) below.

Shared Key: The application provides pre-shared keys as the trust anchors to the trust engine with the parameter `local=true`.

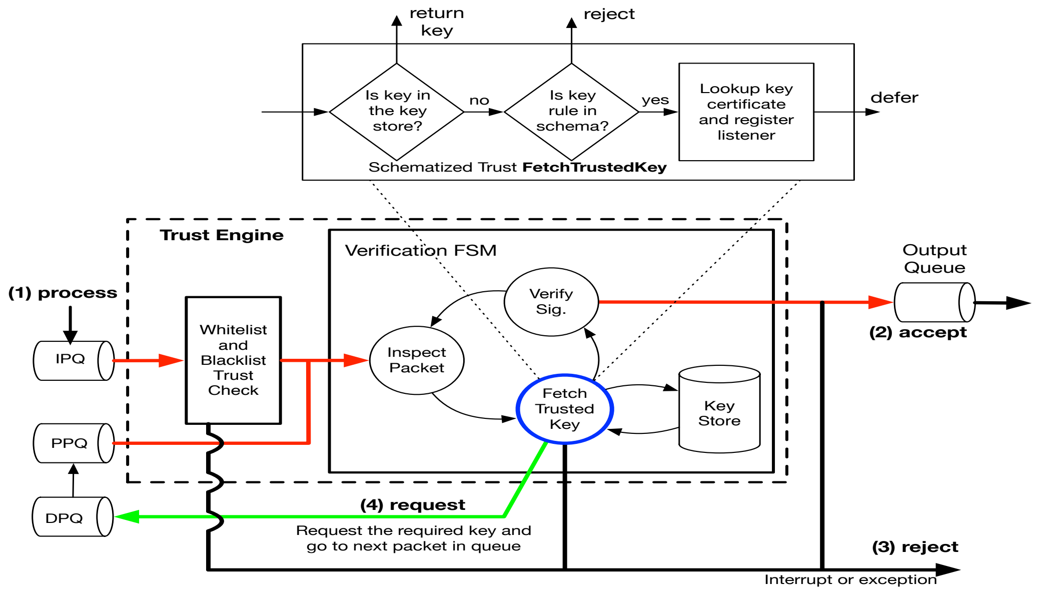

Fig. 8: A portable CCNx trust engine. The implementation only requires an interface to record interests, process (accept or reject) content objects, and fetch additional data (e.g., certificates). The trust engine context is stored internally. A configuration interface (not shown) allows the trust model rules to be modified at runtime.

Subordinated Hierarchy PKI Application Trust: A set of trust anchors (and their public keys) are collated and provided to the trust engine. The application will then provide these keys to the trust engine with the key-fetch parameter `local=false`.

Schematized Trust (Restricted Hierarchy): The application will perform exactly as outlined in the previous case but will also provide a set of schema rules to the trust engine with the key-fetch parameter `schema=(schema)`. The schema is a JSON-encoded string that describes the rules as listed in [20].

PGP-like Web-of-Trust: The application will provide the same set of trust anchors as in the hierarchical case but will additionally provide the confidence threshold and multiplicative degradation factor by which keys lose trust. Specifically, it will provide `WOT-threshold=(numerical value)` and `WOT-reduction=(numerical value)` values which will be used to enforce the verification rules.

VIII. Related Work

There has been substantial past work focusing on the relationship between names and security in ICN architectures. [10] studies the requirements for constructing trust associations between (1) real-world identities (coupled to the Content Object producers), (2) names, and (3) public keys. In the related works [6], [10], [12], a self-certifying naming scheme is proposed for specifying the exact Content Objects which can be returned in response to an interest. The IKB rule from [9] specifies that the hash digest associated with a self-certifying name be specified outside of the name so as to not mix content identification with trust context.

The concept of automated trust verification with hierarchical and schematized trust models were first introduced in [19], [20]. These provide some details about the actual implementation of a trust engine for NDN (Athena). However, they do not formulate different trust models in a cohesive design; the focus is only on that of schematized trust, whereas our trust engine can incorporate several different trust models into the same context for verification. Moreover, the implementation verifies packets one at a time, whereas the CCNx trust engine described here can handle the asynchronous nature of validating multiple (unrelated) Content Objects. This allows more expensive verification procedures to be interleaved as needed. Lastly, the implementation in [19] will only work in NDN; it is not easily portable to other ICN architectures due to its dependence on the NDN naming scheme and packet format.

With regards to attacks exploiting a lack of trust information, [5] and [2] both studied Denial of Service (DoS) attacks (based on producer and router interest flooding) and probabilistic countermeasures. Content poisoning was defined in [7], and a countermeasure based on analyzing interest exclusion[4] patterns for cached content to determine whether it is fake.

IX. Conclusion

This paper has three important contributions. First, it presents abstract trust model templates which can be used to design application-specific trust models that are transparently enforced within the transport stack of ICN-style networks. Second, by showing executable Prolog code, complex trust concepts are spelled out and become tangible for an implementor, also showing pitfalls when applying these concepts.

[4]Interest exclusion fields are no longer supported in CCN.

Third, it provides a detailed description of the design, implementation, and control of a trust engine for CCNx that is responsible for enforcing application-layer trust semantics. Our contributions help bridge the gap from theoretical trust concepts to practical and secure information centric networking. One remaining piece to explore as future work are the APIs for the trust engine. Ideally, these would be agnostic to different trust models but also flexible enough to give the application developer control over trust semantics. Finally, we plan to conduct overhead and throughput experiments with our trust engine under realistic application workloads.

References

[1] A. Abdul-Rahman, "The pgp trust model," in *EDI-Forum: the Journal of Electronic Commerce*, vol. 10, no. 3, 1997, pp. 27–31.

[2] A. Afanasyev, P. Mahadevan, I. Moiseenko, E. Uzun, and L. Zhang, "Interest flooding attack and countermeasures in named data networking," in *Proceedings of the IFIP Networking Conference*, 2013.

[3] M. Baugher, B. Davie, A. Narayanan, and D. Oran, "Self-verifying names for read-only named data," in *Computer Communications Workshops (INFOCOM WKSHPS), 2012 IEEE Conference on*. IEEE, 2012, pp. 274–279.

[4] "Content Centric Networking project (CCNx)," http://www.ccnx.org.

[5] A. Compagno, M. Conti, P. Gasti, and G. Tsudik, "Poseidon: Mitigating interest flooding DDoS attacks in named data networking," in *Proceedings of the 38th IEEE Conference on Local Computer Networks (LCN)*, 2013.

[6] S. K. Fayazbakhsh, Y. Lin, A. Tootoonchian, A. Ghodsi, T. Koponen, B. Maggs, K. Ng, V. Sekar, and S. Shenker, "Less pain, most of the gain: incrementally deployable ICN," in *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*. ACM, 2013, pp. 147–158.

[7] P. Gasti, G. Tsudik, E. Uzun, and L. Zhang, "Dos and ddos in named data networking," in *Computer Communications and Networks (ICCCN), 2013 22nd International Conference on*, July 2013, pp. 1–7.

[8] C. Ghali, G. Tsudik, and E. Uzun, "Needle in a haystack: Mitigating content poisoning in named-data networking," in *Proceedings of NDSS Workshop on Security of Emerging Networking Technologies (SENT)*, 2014.

[9] ——, "Network-layer trust in named-data networking," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 5, pp. 12–19, Oct. 2014. [Online]. Available: http://doi.acm.org/10.1145/2677046.2677049

[10] A. Ghodsi, T. Koponen, J. Rajahalme, P. Sarolahti, and S. Shenker, "Naming in content-oriented architectures," in *Proceedings of the ACM SIGCOMM workshop on Information-centric networking*. ACM, 2011, pp. 1–6.

[11] A. Jøsang, "An algebra for assessing trust in certification chains." in *NDSS*, vol. 99, 1999, p. 6th.

[12] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica, "A data-oriented (and beyond) network architecture," *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 4, pp. 181–192, 2007.

[13] J. Kurihara, C. Wood, and E. Uzuin, "An encryption-based access control framework for content-centric networking," *IFIP Networking 2015*, 2015.

[14] M. Mosko, E. Uzun, and C. Wood, Internet-Draft draft-wood-icnrg-ccnxkeyexchange-01, Accessed: 2016-05-07. [Online]. Available: https://github.com/PARC/ccnx-keyexchange-rfc

[15] "Named Data Networking project (NDN)," http://named-data.org.

[16] I. Solis, "CCN 1.0 (tutorial)," in *ACM ICN 2014*, Sep. 2014.

[17] W. Stallings, "The pgp web of trust," *Byte*, vol. 20, no. 2, pp. 161–162, 1995.

[18] C. Tschudin and C. Wood, "File-Like ICN Collection (FLIC)," Internet Engineering Task Force, Internet-Draft draft-tschudin-icnrg-flic-00, Apr. 2016, work in Progress. [Online]. Available: https://tools.ietf.org/html/draft-tschudin-icnrg-flic-00

[19] Y. Yu, "Athena: A Configurable Validation Framework For NDN Applications."

[20] Y. Yu, A. Afanasyev, D. Clark, V. Jacobson, L. Zhang *et al.*, "Schematizing trust in named data networking," in *Proceedings of the 2nd International Conference on Information-Centric Networking*. ACM, 2015, pp. 177–186.

Appendix

Prolog permits easy formalization and reasoning about the validity of certificates according to some given trust model. The code in Section V (Fig. 1 to 6) expresses in few lines (i) the validation logic for different trust models, including schematized trust. In the lines below we also model in Prolog (ii) a packet parser and simulated signature verification, as well as (iii) an instance of a schematized trust instance which suffers from the described loop problem (see also Figure 7).

```
% Prolog code to demo the loop case for schematized trust

% INSERT HERE all code snippets from Section V

isValidSignature(h(val(Msg)), Signature, val(KeyBits)) :-
  % mimic the computation of a signature: add Msg value to Key value
  S is Msg + KeyBits,
  Signature = s(S).

demoTrustContext(C) :-
  C = trustCtx('schematized',
        [ % trust anchor (public key of Root)
          key(name(['root','key']), _, val(104)) ],
        [ % schema(SignerKeyName, SignedKeyName)
          schema(name(['root','key']), name(['ping','key'])),
          schema(name(['pong','key']), name(['ping','key'])), % CAUSE 1!
          schema(name(['ping','key']), name(['pong','key'])),
          schema(name(['pong','key']), name(['pkt',_]))
        ]).

% two certificates for the same key of Ping:

ccnFetchCert(N, pkt(N, val(Msg), KeyInfo, h(val(Msg)), s(S))) :-
  N = name(['ping','key']),
  Msg = 103,
  KeyInfo = key(name(['pong','key']), id(102), _),
  S = 205.

ccnFetchCert(N, pkt(N, val(Msg), KeyInfo, h(val(Msg)), s(S))) :-
  % CAUSE 2: Root's certificate is fetched AFTER Pong's cert (above)
  N = name(['ping','key']),
  Msg = 103,
  KeyInfo = key(name(['root','key']), id(104), _),
  S = 207.

% one certificate for Pong's key:

ccnFetchCert(N, pkt(N, val(Msg), KeyInfo, h(val(Msg)), s(S))) :-
  N = name(['pong','key']),
  Msg = 102,
  KeyInfo = key(name(['ping','key']), id(103), _),
  S = 205.

loopDemo :-
  % this predicate will loop forever (= stack overflow)
  % how to break: remove the one schema line, or change order of certs,
  % or fetch certs by KeyId, which requires changes to ccnFetchCert()
  Msg = val(1000),
  Pkt = pkt(name(['pkt','678']), Msg,
            key(name(['pong','key']), id(102), _),
            h(Msg), s(1102)),
  demoTrustContext(Ctx),
  isValidPkt(Pkt, Ctx, _).

% data structures used:
%
% pkt(name(CompList), val(Msg), KeyInfo, h(Data), s(Signature))
%
% key(name(CompList), id(KeyId), val(Bits))
%
% trustCtx(TrustModel, ListOfTrustedKeys, Aux)
%   where Aux is either SchemaList, ConfidenceList, or _
%
% schema(SignerKeyName, SignedKeyName)
%
% confid(FriendName, Float)
```