NDN, Technical Report NDN-0030, 2015. http://named-data.net/techreports.html
Revision 1: April 20, 2015

Athena: A Configurable Validation Framework For NDN Applications

Yingdi Yu
UCLA
yingdi@cs.ucla.edu

ABSTRACT

In Named Data Networking (NDN), data is bound with its name through its producer's public key signature. Secure communication in NDN requires every piece of data to be authenticated. Leaving the data authentication to application developers is error-prone. It is more reasonable to ask application developer to select a pre-defined trust model, and have a security library to automatically set up all the data authentication procedures. In this paper, we proposed Athena, a configurable validation framework to support automated data authentication. We also defined a policy language Guardian to help security experts to specify a variety of trust model in a convenient way.

1. INTRODUCTION

Named Data Networking (NDN) is designed with built-in security. The name and content of a data packet are bound together with a signature. The process of packet validation, however, may not be trivial.

First of all, one must determine whether the signer of a packet can be trusted for signing the packet; and one also needs to determine that the signing key actually belongs to the trusted signer. The approach of making the two decisions above may vary from one application to another application, depending on the trust model of the application.

Second, in order to verify a signature, one may need to retrieve the corresponding public key and related authentication information, such as certificates issued by some third parties, the signature status, etc.. The information to retrieve depends on the security requirements of an application. For example, some applications using web-of-trust may want to fetch all the endorsement certificates for a key from a particular key server, while some other applications may want to retrieve certificates directly. Moreover, constrained by the network environments in which an applications is running, the mechanisms to retrieve authentication information could also be different. For example, when the network access of a data consumer is limited to a data producer only, the data consumer may want to retrieve all the authentication information from the producer rather than sending interests to other parties which will never respond.

Third, the signature validation process also requires some careful checkings. For example, one may need to check whether the signature has been revoked before its validity period ends; or one may need to check whether the issuer of a certificate is allowed to delegate some trust to others.

It is unnecessary and error-prone to leave all the implementation details to NDN application developers, while developers should still be allowed to experiment a variety of trust models and other security features in the validation process. Therefore, it would be desired to provide a modularized framework for packet validation, which handles as many common validation procedures as possible but is still flexible enough for developers to extend.

In this paper, we first identify the requirements of packet validation in NDN applications, and generalize a common validation procedure. Based on that, we proposes `Athena`, a modularized framework of packet validation in NDN applications.

The rest of the paper is organized as follows. In Section 2, we briefly introduce the security semantics of NDN. The requirements of packet validation are examined in Section 3. The `Athena` framwork is presented in Section 4 and we further elaborate two modules of `Athena` in Section 5 and 6 respectively. We demostrate the framework with two existing examples in Section 7 Some other related issues are discussed in Section 8. We conclude the paper in Section 9.

2. NDN SECURITY

NDN Security is data-oriented. As shown in Figure 1, the content and its name are bound together through a digital signature, along with the meta information of the content and the signature. The meta information of the signature includes the signature type (`SignatureType`), the signer (`KeyLocator`), the validity period of the signature `ValidityPeriod`, and may also include extensions such as signature status, etc..

Ideally, every NDN data packet must be validated be-

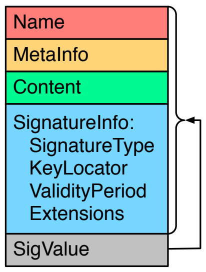

Figure 1: The format of an NDN data packet.

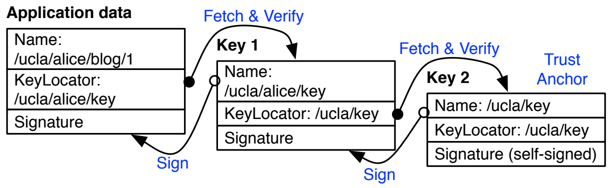

Figure 2: An example of data validation in NDN. The last key is pre-trusted and is represented as a self-signed key.

fore being used. In order to validate a data packet, a data consumer must retrieve the correct public key to verify the data signature. A public key, as a special type of content, is also named and can be retrieved as a data packet in NDN. The producer of a data packet is supposed to provide the retrieval information about the public key in the `KeyLocator`, for example, the name of the public key. Since every data packet is signed in NDN, a data packet containing a public key is essentially a public key certificate whose issuer is the signer of the packet. Thus, a certificate can be validated in the same way as a normal data packet in NDN. Figure 2 shows an example of data validation in NDN.

A verified signature does not necessarily means the data packet is valid. A data consumer must also decide whether the data signer can be trusted for signing a particular data packet. The name spaces for which a key can be trusted is defined as the trust scope of the key. Given the same public key, its trust scope could be different in different applications. The specification of the trust scope of public keys and some other conditions on the signer (such as the signature type) constitutes the policy of an application.

Moreover, a signature may have a limited lifetime for the intential restriction on the effect of the name-content binding or for the cautiousness that the signing algorithm may be broken eventually. Therefore, a data packet is usually treated as valid by a data consumer if 1) it complies with the application's policy; 2) its signature can be verified; 3) its signature has not expired yet.

3. VALIDATION REQUIREMENTS IN NDN

In this section, we identify the requirements of packet validation in NDN on three aspects: *policy checking*, *key retrieval*, and *signature verification*.

3.1 Policy Checking

On receiving a packet, one should first check if the packet complies with the policy, because: 1) it is the only procedure that does not require any other information except the packer per se and 2) retrieving other information could be complicated and costy.

The policy checking is focused on the `SignatureInfo` of a packet. For each sub-field in `SignatureInfo`, one may specify the required condition. These conditions should describe the trusted keys of the packet, the acceptable signature types, etc.. A packet will be treated as invalid immediately if one of the sub-fields cannot satisfy its corresponding condition. For different packets, the conditions on `SignatureInfo` could be different. We generally call the conditions for all the packets that an application expects to validate as *Policy*.

Note that the checking procedure for the same sub-field does not change among applications. It would be unnecessary and error-prone to ask application developers to implement these procedures on their own. Therefore, *a desired validation framework should provide standard checking procedures for each sub-field of SignatureInfo*.

Although the checking procedure can be standardized, the policy or the conditions may vary a lot from one application to another application. For example, the condition on `KeyLocator` (one of the `SignatureInfo` sub-fields), which restricts the trusted keys of a packet, actually defines the validation trust model of an application and may be highly customized. Thus, *a desired validation framework should provide enough flexibility in specifying policies.*

3.2 Public Key Retrieval

If a packet has passed the policy checking, the next step is to get the public key to verify the signature. One should first look up the public key in the trust anchors. Trust anchors are a set of public keys that are pre-authenticated before any validation process begins. Therefore, a trust anchor can be directly used to verify the signature[1] without requiring other information. This is also why the lookup should be done before looking for the public key in other places. *A desired validation framework should provide efficient trust anchor*

[1]This does not mean that any packet signed by a trust anchor will be treated as valid, because the policy checking step should filter out the packets that are out of the trust scope of the trust anchor.

lookup and management.

If the public key is not in the trust anchors, one may need to retrieve the public key from some other places. The key retrieval mechanim may be different from one application to another application. For example, some applications may assume that the data provider will also provide the public key to verify the signature, thus they may send an interest towards the data provider. However, some other applications may assume that the public key can be retrieved using the key name directly regardless of who actually serve the key. In some cases, one does not need to retrieve a public key, but also some information associated with the key, such as endorsements, delegations, etc.. Therefore, *a desired validation framework should support flexible key retrieval mechanism.*

Since a public key is also retrieved as a data packet in NDN, it can be validated in the same framework. As a result, a validation process may develop recursively until a trust anchor is reached. A loop detection mechanism must be provided and some step constraints should also be enforced to avoid infinite incursions of public key retrieval. When a public key is required to be certified by multiple parties, the validation process may even fork into multiple branches. Thus, *a desired validation framework should build the validation paths correctly and effectively.*

3.3 Signature Verification

When the validation process reaches a trust anchor or an intermediate public key has been authenticated, the public key can be used to verify the corresponding signature. Although signature verification is trivial with a correct public key, the procedure after signature verification could be complicated.

When the validated packet contains some intermediate authentication information, the content of the packet must be parsed appropriately, otherwise the validation process cannot proceed. Although the content format of many intermediate authentication data (such as certificate) is defined as a public standard, some applications may require certain authentication information that is encoded in a private format.

Moreover, some application may require checking the status of the signature in case the signature has been revoked before its validity period ends. Signature status checking should be done after the signature verification, because the signature status checking could be expensive and it is worthless to check the status of an invalid signature. However, the signature status checking mechanism has not been defined in the NDN architecture yet. Therefore *a desired validation framework should provide an interface for users to extend the post-verification data processing.*

3.4 Validation Result Processing

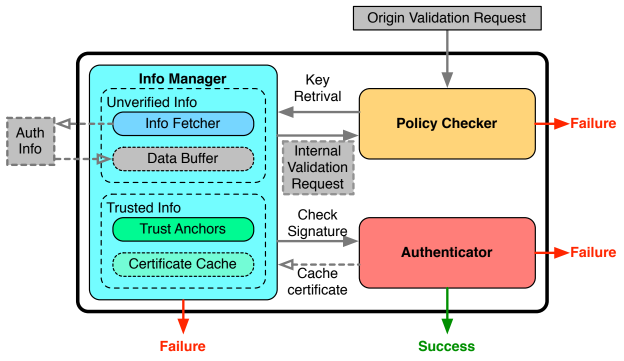

Figure 3: The validation framework

Although there is only one exit for a successful validation, a validation process may fail for a variety of reasons. Some failures are recoverable. Some failures may help application to adjust its behaviors. It is also possible that the same failure would be treated in different ways in different applications. Therefore, *a desired validation framework should allow applications to handle the signature verification result in their own ways.*

4. VALIDATION FRAMEWORK

The analysis above identified the requirements of packet validation in NDN, and also suggested that the process of packet validation can be modularized. In this section, we introduce Athena, a modularized framework of packet validation in NDN applications. As shown in Figure 3, the framework consists of three modules: PolicyChecker, InfoManager, and Authenticator. Each module corresponds to one aspect that we analyzed before In the rest of this section, we will first introduce the interface, working flow, and each module of the framework, and then discuss two the validation process examples.

4.1 Framework I/O

The input of Athena contains the packet to validate and some other meta information, such as the procedure to handle validation result. The input is expressed in terms of a data structure called *validation request*. As we will see later, the same data structure is also used for some intermeidate authentication information (such as public keys) that are fetched and validated internally by the framework. For the sake of clarity, the two types of validation request are called *original validation request* and *internal validation request* respectively.

A validation request contains following information:

- `ProcessId`: the identifier of a validation process.
- `Target`: the packet to validate.

- SuccessCallback: a callback function when Target is validated.
- FailureCallback: a callback funciton when Target cannot be validated.
- MaxSteps: the maxium number of validation steps that can be performed for this request.

Since a validation process is all about the target packet in the original validation request, ProcessId is set to be the digest of the original packet. Note that, there could by more than one validation requests sharing the same ProcessId, because the framework may generate some internal validation requests for intermediate authentication information, which are eventually used to validate the target of the original request. However, within the same validation process, each individual validation request is uniquely identified by its Target.

For each validation request, there are two callback functions that will be invoked according to the validation result. The callback functions of the original validation request should be supplied by application in order to satisfy the requirement of validation result handling in Section 3.4. Note that the validation process may fail for different reasons. Some validation failures (such as "cannot retrieve the public key") does not necessarily mean the target packet is invalid. Therefore, when the FailureCallback is invoked, a failure code will also be passed as a parameter, so that application can handle various validation failures in the FailureCallback on their own.

Indeed, a validation process starts when the original validation request is submitted to the framework and ends when either SuccessCallback or FailureCallback for the original validation request is invoked. The callback functions for internal validation requests connect together the validation steps in a single validation process. We will discuss the callback functions and MaxSteps in detail later.

4.2 Asynchronized Validation

In order to validate the target packet in the original validation request, an application should retrieve the corresponding public key for the signature verification. The retrieved public key cannot be used if it has not been authenticated. Thus, an unauthenticated public key (carried by a data packet) will be passed as an internal validation request back to PolicyChecker. An internal validation request may trigger another internal validation request if the signing key of the target of the first internal validation request has not be authenticated neither. Such a process will develop recursively until a pre-authenticated key is reached. Once an intermediate key has been authenticated, it can be used to verify the signature of depending packets (or certificates). In Athena, such a validation process is modeled as an asynchornized process and is achieved by a set of callback functions that are chained together. Figure 4 shows an example of the asynchornized validation.

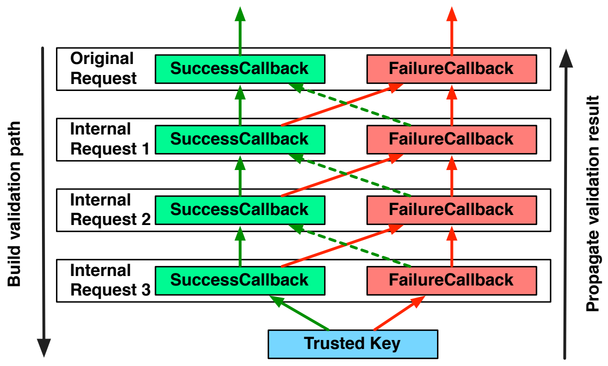

Figure 4: An example of asynchronized validation process

For simplicity, we assume that each internal validation request in Figure 4 is for the signing key of the target of its previous validation request. In Athena a validation path is first recursively built till an authenticated public key, and weaves the dependency between the callback functions of the internal validation requests.

When the Target of the last internal validation request (the third one in Figure 4) is validated, the corresponding SuccessCallback will be invoked. In this callback, the authenticated public key is used to validate the depending request (the second one in Figure 4). If the target of all the subsequent validation request can be validated, the success will be propagated through the SuccessCallback to the one of the original validation request. If an intermediate target cannot be validated, the corresponding FailureCallback will be invoked. In most cases, the failure will be propagated through the FailureCallback to the one of the original validation request.

Note that FailureCallback may not be only invoked when signature verification fails. If the failure is recoverable (e.g., temporary connectivity issue), the FailureCallback can still get the validation path back onto the desired track, as the dashed lines suggest in Figure 4.

Since the validation path is built recursively, it is important to control the depth of the whole recursion. Although in most cases a well-defined policy can prevent infinite recursion by restricting the trusted signing keys, a hard limit is still needed in case of errors in the policy. The MaxSteps in a valiation request is designed for this purpose. In Athena, the MaxSteps of each internal validation request is required to be less than the one of its previous validation request. As we will see in Section 4.4, a request with non-positive MaxSteps will be rejected directly.

4.3 PolicyChecker

PolicyChecker, as its name suggests, is the module to check packet against policy. As we discussed in Section 3.1, the policy checking should be done first. Therefore, the PolicyChecker module is the first module to handle any validation request in Athena.

Recall that we identified two requirements for policy checking in Section 3.1: 1) provide standard checking procedures and 2) allow as much flexibility as possible in policy specification. The Athena framework fulfills the two requirements by providing a configurable PolicyChecker module.

For each sub-field of `SignatureInfo` that is defined in NDN packet format, PolicyChecker provides a built-in checking procedure with configurable conditions. Application can customize the PolicyChecker module by

- selecting a set of checking procedures for a group of packets that share the same functionality
- specifying the condition for each selected checking procedure

Such customization is done by writing a configuration file using a policy language called Guardian. As we will show in Section 5, Guardian can be used to describe a variety of conditions on `SignatureInfo`, thus giving applications sufficient flexibility to specify their own policies.

PolicyChecker is initialized with a configuration. All the policy checking procedures are constructed automatically according to the configuration. When PolicyChecker receives a validation request, a corresponding set of checking procedures will be selected to inspect the `SignatureInfo` of the target packet. If the target packet can pass all the checking, the validation request will be delivered to the InfoManager module, which will be introduced later. Otherwise, the `FailureCallback` will be invoked to terminate the validation process.

The benefit of a configurable PolicyChecker is obvious. The configuration file provides an convenient interface for an application to specify their own policy without worrying about implementation details. As a result, the users of the Athena framework can easily experiment different trust models by simply changing the PolicyChecker configuration.

4.4 InfoManager

The InfoManager module is designed to handle the public key retrieval which is discussed in Section 3.2. When necessary, the module may also retrieve other authentication related data, such as the status of a signature, endorsement, delegation, etc.. We generally call all these data as *authentication information*. Beside authentication information retrieval, InfoManager also provide adequate management of these information.

Recall that we identified three requirements regarding authentication information management in Section 3.2: 1) efficient trust anchor lookup and management, 2) flexible information retrieval mechanism, and 3) effective validation path construction. The InfoManager satisfies the three requirements by providing two sub-modules: TrustAnchors and InfoFetcher. Besides the two basic sub-modules, there are two more optional sub-modules in InfoManager: CertificateCache and DataBuffer, which are designed to improved the performance of InfoManager. In order to provide sufficient flexibility, all the four sub-modules are configurable and extensible. We will briefly introduce the four sub-modules here and discuss the design details to Section 6.

When a validation request is delivered to InfoManager, it will be first processed by TrustAnchors. The TrustAnchors sub-module looks up the public key that matches the `KeyLocator` of the target packet. If there is a match, the validation request together with the public key will be delivered to the Authenticator module, which will be introduced later.

If the signing key of the target packet is not a trust anchor, the validation request will be processed by CertificateCache if it is enabled. The CertificateCache sub-module can cache intermediate public key certificates that have been authenticated in some other previous validation processes, thus avoiding unnecessarily public key retrieval and authentication. The existence of CertificateCache may significantly improve the performance of the validation framework, especially in terms of latency.

If the signing key cannot be found from neither TrustAnchors nor CertificateCache, the validation request will be processed by InfoFetcher. InfoFetcher will fetch the public key from the network according to `KeyLocator` of the target packet. It may also fetch additional authentication information when necessary.

InfoFetcher needs to perform two additional procedures before fetching any requested data. First, it must check `MaxSteps` to determine whether the data fetching is allowed. Second, it must check whether the interest for the same data has been sent out before in the same validation process (as identified by `ProcessId`). The fetched data packets cannot be used without validation. Therefore, InfoFetcher will prepare an internal validation request for the fetched data and deliver the validation request to PolicyChecker, so that the intermediate public key can be validated in the same way as the original target packet. To indicate that the new validation request consumes one further step, the `MaxSteps` in the new validation request will be decreased by 1 from one of the previous validation request. All the procedure above effectively enable loop detection and avoid infinite validation path in Athena.

In some cases, InfoFetcher may bring back more authentication information that requested or application may supply some addtional authentication information along with the original validation request. Although these information may not be used immediately to validate for the original target packet, they could be useful in validaing some intermediate public keys. The DataBuffer sub-module is designed to hold these information temporarily to avoid unnecessary data fetching. Similar as CertificateCache, DataBuffer is also designed for performance optimization, thus being optional. When DataBuffer is enabled, InfoFetcher will look for the requested data in DataBuffer before sending out interests to the network.

4.5 Authenticator

The last module is Authenticator which is designed to handle signature-related processing as discussed in Section 3.3 and trigger corresponding callback functions. Since the validation request is always delivered to Authenticator with an authenticated public key, the signature of the target packet can be easily verified in the Authenticator.

As we identified in Section 3.3, a desired validation framework should allow users to extend the post-verification data processing. The Authenticator module satisfies such a requirement by provding a processing callback system. For any data packet that requires further processing after signature verification, a corresponding processing callback can be registered in the Authenticator module. When the signature of a packet has been verified, the packet will be matched against all the registered callbacks and will be processed by the matched callback function.

5. GUARDIAN

As we mentioned in Section 4, the core part of the PolicyChecker module is the policy configuration. The configuration must be clearly specified, so that the conditions on `SignatureInfo` can be accurately interpreted by PolicyChecker. Therefore, we defined Guardian, a policy language for configuration specification.

Guardian is designed with two goals. First, it should facilitate users to classify packets for different checking procedure sets. Second, it should allow users to easily specify conditions on each `SignatureInfo` sub-field. As a result, Guardian is designed to provide a *rule-based* policy configuration. A policy written in Guardian consists of a list of *rules*. Table ?? shows an example configuration. Each rule corresponds to a particular group of packets that share the same checking procedure set and contains the conditions for each checking procedure in the set. A packet has to pass a rule in the policy in order to pass the PolicyChecker module.

A rule consists of two parts: a *filter* and a set of *checkers*. A checker defines the conditions for a particular checking procedure set, while the filter qualifies the packets to which the checkers should be applied. A packet can pass a rule only if the packet is matched by the filter and can pass one of the checkers in the rule. A rule usually contains only one checker. However, when there could be more than one sets of valid conditions on for the matched packets, a rule may contain more than one checkers.

```
rule {
 filter {
  packet-type data
  packet-name <localhost><secured><>*
 }
 checker {
  signature-type rsa-sha256
  min-key-size 2048
  key-locator {
   name <localhost><admin><KEY><><ID-CERT>
  }
 }
 checker {
  signature-type ecdsa-sha256
  min-key-size 256
  key-locator {
   name <localhost><admin><KEY><><ID-CERT>
  }
 }
}
rule {
 filter {
  packet-type data
  packet-name <localhost><unsecured><>*
 }
 checker {
  signature-type sha256
 }
}
```

Figure 5: An example of a policy written in Guardian.

Rules in a policy are organized as a list. A packet will be matched against the filter of each rule from the beginning of the list until the first rule whose filter matches the packet. Only the checkers of the first matched rule will be applied to the packet. In other word, once a packet is matched by a rule, the packet will either pass the policy checking or be treated as invalid immediately according to the checking result of the matched rule, regardless of whether the packet can pass some other rule later or not. As a result, the order of the rules matters in a policy. A rule with a more specific filter should be placed before those with a less specific filter. If a packet cannot be matched by any rule in a policy, the packet will be treated as invalid immediately. In the rest of this section, we will explain *filter* and *checker* in detail.

5.1 Name Pattern

Before we go into details about filter and checker, it would be helpful to introduce an important concept in Guardian: *name pattern*, because it is inevitable, within the NDN context, to specify the conditions on name in both filters and checkers. For example, a filter may match a packet according to the packet name, while the name in `KeyLocator` of a packet may have to satisfy certain conditions.

Name pattern is a tool to describe the conditions on NDN names. It resembles regular expression and provides two functionalities: 1) match an NDN name, and 2) extract information out of an NDN name. The second functionality is particularly useful in expressing the condition on the relationship between the name of a packet and the name of the packet signer.

A name pattern is constructed at two levels: component level and name level. Each level corresponds to a different matching scheme. At the component level, only a single name component is matched. The component level matcher is expressed as a regular expression enclosed with a pair of "<" and ">". For example `<ucla>` can match a name component `ucla`. The only exception is "`<>`" which is a wildcard matcher that can match any single name component.

At the name level, the pattern of name components is matched. The name level matcher describes the order of name components and their repetitions. For example, a pattern matcher `<edu><ucla><cs>` can match a name `/edu/ucla/cs`. We also define several meta-characters which resemble those in the standard regular expression:

- `(`, `)`: defines a sub-pattern which can be referred using `\n`.
- `+`: one or more repetition of the preceding component or sub-pattern.
- `*`: zero or more repetition of the preceding component or sub-pattern.
- `{n, m}`, `{n,}`, `{, m}`, `{n}`: customized repetition of the preceding component or sub-pattern.

With these meta-characters, one can specify for complicated name patterns For example, a name pattern `<edu><ucla><>*` can match any name with the prefix `/edu/ucla`. A name pattern `(<>*)<DNS>(<>*)<NS>` with back reference `\1\2` can extract the domain name of an NDNS name server record (e.g. `/edu/ucla/cs` will be extracted from `/edu/ucla/DNS/cs/NS`). As we will see later, such component extraction can help applications to effciently specify their trust models.

5.2 Filter

Filter is used to classify packets according to their properties, so that packets can be checked through appropriate procedures. For each property specified in a filter, there is a value that a qualified packet must match. In other word, given a filter, a qualified packet must match all the specified property value. For now, Guardian supports two packet properties in filter: 1) packet-type and 2) packet-name.

The packet-type property has only two possible values: data and interest[2]. This property must be supported because the checking procedures for a data packet and a signed interest could be different, even though the packet name could be the same.

The value of the packet-name property is a name pattern that can match the packet name. This property must be supported because the name of a packet contains most information about a packet, such as the description of the content (for a data packet) or the usage of a command (for a signed interest).

One does not have to specify all the properties in a filter. If the value for a property is missing, it will be interpreted as "any" and PolicyChecker will skip matching the property.

Filter can be extended to refine the packet classification by introducing new packet property types. Note that introducing new property types does not affect the correctness of old filter specification, thus the extension on filter is always backward compatible.

5.3 Checker

Checker is used to inspect the `SignatureInfo` of a packet. Since two sub-fields (`SignatureType` and `KeyLocator`) are defined in `SigInfo` by the NDN specification [1], current checker implementation supports both of them.

The condition on `SignatureType` should be one of the signature type defined in the NDN specification [1]. This condition represents the security level on a particular group of packet in a policy. For example, in the policy in Table 5, the `SignatureType` in the first rule is required to be a strong signature type (such as `rsa-sha256`), the security requirements to the data packets matched by this rule is apparently higher than those matched by the second rule in which a `sha-256` digest is sufficient enough.

Compared to the condition on `SignatureType`, the condition on `KeyLocator` could be more complicated. Although in some cases one can specify the legitimate signing key name directly (such as the first rule in Table 5), in some other cases, the name of a legitimate signing key depends on the name of the packet. For example, in the hierarchical trust model, a packet must be signed with a key that represents one of the packet's parent name spaces. In order to describe such a dependency, Guardian provides a new type of condition, called *hyper-relation*, which leverages the component extraction functionality of name pattern.

Hyper-relation is defined with three pieces of information:

- `p-pattern`: name pattern for packet name. With

[2]See more details about signed interest at `http://named-data.net/doc/ndn-cxx/current/tutorials/signed-interest.html`

```
key-locator {
 k-pattern (<>*)<KEY><><ID-CERT> \1
 h-relation is-prefix-of
 p-pattern (<>*) \1
}
```

Figure 6: An example of hyper-relation.

```
key-locator {
 k-pattern (<>*)<admin><><KEY><><ID-CERT> \1
 h-relation equal
 p-pattern (<>*)<router><><KEY><><ID-CERT> \1
}
```

Figure 7: An example of hyper-relation for non-hierarchial trust model.

p-pattern components will be matched from packet name and expanded into a new name space N_p.

- `k-pattern`: name pattern for `KeyLocator`. Components from `KeyLocator` will be matched and expanded into another new name space N_k.
- `h-relation`: relation from the signer's expanded name space N_k to the packet's expanded namesspace two new name spaces N_p. Its value may be "`equal`", "`is-prefix-of`" and "`is-strict-prefix-of`".

Table 6 shows an example of hyper-relation.

Since hyper-relations regulate the relation between a packet name and its signer name, they actually define the trust model of an application. For example, the hyper-relation in Table 6 defines a hierarchical trust model, i.e., a key is only trusted to sign data under its own name space.

Hyper-relations can also be used to define non-hierarchical trust model. Table 7 shows such an example. In this example, an administrator is trusted to sign the key of routers in the same network. Although the key name of a router `/ucla/router/1` is not under the name space of the administrator key `/ucla/admin/alice/`, the hyper-relation in Table 7 can extract the site prefix from both names and compare them. Therefore, the trust can be derived from the administrator of a site to a router in the site's network.

Besides supporting the fields that are explicitly defined, checker may also check some implicit information about signature, e.g., key size. The key size is particularly important, because for some signature algorithms the key size determines the strength of signature. As a result, one can specify the minimum key size in checker through the `min-key-size`. For example, in Table 5, the size of an RSA key is required to be at least 2048.

5.3.1 *multiple checkers*

There are might be several sets of valid conditions on the same group of packets. For example, for one catergory of contents, both RSA signatures and ECDSA signatures are accepted and the minimum key size may be different for each signature type. As another example, it is possible that more than one signers (with different identities) are trusted for signing the same content. Guardian supports multiple condition sets by allowing multiple checkers in a rule. A packet only needs to pass one of the checkers.

6. INFO MANAGER

As we mentioned in Section 4.4, InfoManager prepares the public key that may be used to verify the packet signature. The module consists of two sub-modules: TrustAnchors and InfoFetcher and may inlcude two optional caching sub-modules: DataBuffer and CertificateCache. All these sub-modules are configurable. In the rest of this section, we will discuss these sub-modules in detail.

6.1 TrustAnchors

The TrustAnchors sub-module stores and manages pre-authenticated keys. Pre-authenticated keys are loaded into TrustAnchors when InfoManager is initialized. Before being loaded, a pre-authenticated key could be stored in a variety of formats. Therefore, TrustAnchors must support a variety of key loading mechanisms.

In some cases, a pre-authenticated key may be replaced by a new key. Such a key rollover should not interupt the operation of the depending application. Therefore, TrustAnchors should also handle key rollover smoothly during the runtime.

The flexibility of loading keys is enabled by the configuration of TrustAnchors. At least three key loading mechanisms are supported so far: *raw*, *file*, and *directory*, as shown in the first three examples in Table 8. When "raw" is specified, the value is the raw public key bits that are encoded appropriately (e.g., base64). When "file" is specified, the value is a path to a file where the public key bits is stored. When "dir" is specified, the value is a path to a directory under which each file contains a public key. When more loading mechanisms are needed, the new mechanism can be supported by introducing more configuration options. Although there is only one TrustAnchors sub-module, multiple `trust-anchor` entries can be specified and all the keys referenced by these entries are loaded into the same TrustAnchors.

Athena provides two runtime key rollover mechanisms. The first one is a programming interface through which application can trigger key rollover on demand. The second mechanism is relative passive and is achieved through a monitoring system. The monitoring system watches for any changes on the sources specified in the

```
trust-anchor {
 raw-base64 "BvODGwdG...amHFvHIMDw=="
}
trust-anchor {
 file /usr/local/ndn/trusted.key
}
trust-anchor {
 dir /usr/local/ndn/keys
}
```

Figure 8: Examples of trust anchor configuration.

```
trust-anchor {
 file /usr/local/ndn/trusted.key
 refresh 1h ; reload the key every hour
}
```

Figure 9: An example of runtime trust anchor rollover configuration.

configuration and adjust the pre-authenticated key set accordingly. User can enable the the watching system by adding a new entry `refresh` in the `trust-anchor` configuration. The value of this entry is the period for which the sources are polled. Table 9 shows an exmpale of enabling the monitoring system.

6.2 InfoFetcher

When the public key for a packet is not one of the trust anchors, Athena may fetch the public key from the network. As we mentioned in Section 3.2, how to fetch a public key depends on the network environment, the application trust model, etc.. In some cases, the same information may be served in different ways, thus the corresponding fetching mechanism could be different as well. For example, a public key may be served as an NDNS record or as a part of a jumbo data that contains all the proofs to authenticate the key. Therefore, it would be desired if InfoFetcher can provide fine granularity control over the data fetching. Moreover, since public keys are critical in data validation, InfoFetcher should also provide sufficient redundancy in data fetching, in case one of the fetching mechanism fails.

All these requirements can be satisfied by the InfoFetcher configuration. As shown in Table 10, the configuration consists of a list of `fetch` entries. Each entry contains a `info-name` field which is a name pattern that can match the name of the requested data and can be expanded to a new name for the information to fetch. Similar to the rules defined in the policy section, the name pf the requested data will be matched from the first entry in the list until a matched entry is found. However, if no entry matches the requested data name, the data name will be directly set as the fetching interest name.

Besides the `info-name`, an `fetch` entry may contains one or more `method` fields which indicate the method

```
retrieval {
 info-name <localhop><register><>*
 method bundle
}
retrieval {
 info-name (<>*) \textbackslash 1<endorsement>
 method ndns
 method bundle
}
```

Figure 10: Examples of InfoRetriever configuration.

```
certificate-cache {
 cache-size 1000
 cache-algorithm lru
 max-freshness 1h
}
```

Figure 11: Examples of CertificateCache configuration.

to fetch the data. The methods are ordered according to their priorities. When the one method fails, the next method will be tried if it exists. As a result, the order of methods represents the strategy of data fetching.

6.3 Caching

As we mentioned before, there are two optional caching sub-modules in InfoManager: DataBuffer for unverified packets and CertificateCache for authenticated packets. Due to their functionality, the configuration of these two sub-modules only specifies the basic properties of a cache including: `cache-size`, `cache-algorithm`, and `max-freshness`. The last property `max-freshness` is specified to prevent a non-stale packet from staying in the cache forever. Table 11 shows an configuration example for CertificateCache. The optional modules will not be enable if the corresponding configuration does not exist.

7. DEMOSTRATION

In this section, we demonstrate two examples of validaiton process that resembles two existing validaiton modes: DNSSEC (DNS Security Extenstions) and TLS (Transport Layer Security).

In DNSSEC mode, authentication information (such as DNSKEY records and DS records) are retrieved separately and can be cached at the validator side for future usage. In TLS mode, all the authentication information[3] are all provided by the signer of the target packet.

7.1 DNSSEC-style validation

In order to support the DNNSEC-style validation, the CertificateCache is enabled while DataBuffer is disabled. Figure 12 shows an example configuration for the DNSSEC-style validation. The policy in the config-

[3]we assume that revocation information is conveyed in terms of OCSP-stapling

```
policy {
 rule {
  filter {
   packet-type data
   packet-name <>*
  }
  checker {
   signature-type rsa-sha256
   min-key-size 2048
   key-locator {
    k-pattern (<>*)<KEY><><ID-CERT> \1
    h-relation is-prefix-of
    p-pattern (<>*) \1
   }
  }
 }
}
trust-anchor {
 raw-base64 "BvODGwdG...amHFvHIMDw=="
}
certificate-cache {
 cache-size 1000
 cache-algorithm lru
 max-freshness 1h
}
```

Figure 12: An example of DNSSEC-style validation configuration.

uration contains a single rule which specifies a hierarchical trust model. The trust anchor is the DNS root expressed in the raw format. The capacity of the certificate cache is set to 1000. The caching strategy is set to Least Recently Used (LRU) and the default TTL is set to 1 hour.

The process flow of DNSSEC-style validation is shown in Figure 13. A target packet is first processed by the PolicyChecker module (step 1). PolicyChecker, according to what is specified in the policy, requests InfoManager for the public key to verify the target packet (step 2). InfoManager then looks up the requested certificate in TrustAnchors and CertificateCache. If the requested certificate is found, InfoManager will deliver the validation request to Authenticator for signature verification (step 7).

If the requested certificate is absent in both TrustAnchors and CertificateCache, the InfoFetcher will be invoked to fetch the certificate (step 3 & 4). The fetched certificate will be passed back to PolicyChecker as an internal validation request (step 5 & 6). The process may develop recursively until one of the authenticated public key is reached (step 2-6). Once the authenticator has validated an intermediate public key, the key can be cached in the CertificateCache (step 8). Depending on the validation result, either `SuccessCallback` or `FailureCallback` will be invoked eventually.

7.2 TLS-style validation

In order to support a TLS-style validation, DataBuffer is enabled to cache all the authentication information

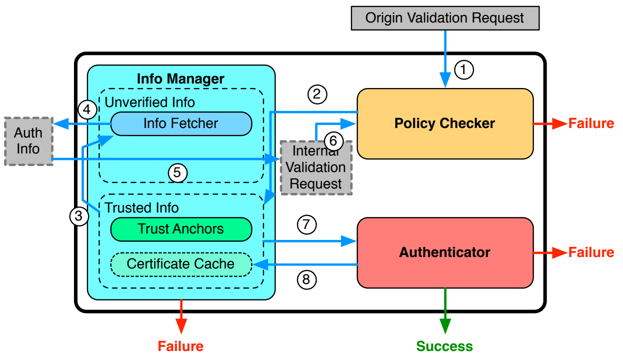

Figure 13: The validation process in DNSSEC mode.

for the signing key. Since all the authentication information has already been supplied, there is no need to enable CertificateCache. Assume the TLS trust model can be expressed through NDN name, e.g., the name of all certificate authority (CA) starts with a name component "CA", an example configuration is shown in Figure 14. In this configuration, the policy contains two rules. The first rule restricts the issuer of a certificate to CAs. The second rule restricts the legtimiate signer of normal data to be the owner of one of the data's parent name spaces. The trust anchor in the TLS mode are root CAs, whose certificates are stored in a directory "`/usr/local/root-ca/keys`". The capacity of data buffer is set to 1000, while the caching strategy of data buffer is set to First In First Out (FIFO).

The process flow of TLS-style validation is shown in Figure 15. When the original traget packet is submitted to the PolicyChecker module, the related authentication information is also cached in DataBuffer (step 1). If the original packet can pass the PolicyChecker, the InfoManager will look up the signer's public key in TrustAnchors (step 2). If the corresponding public key is not a trust anchor, InfoManager then looks for the public key certificate in DataBuffer (step 3) and generates an internal validation request to PolicyChecker (step 4). Since all the authentication information are cached in DataBuffer, the validation process will proceed along with the request exchange between DataBuffer and PolicyChecker (step 2-5), until one of the trust anchor is reached. After that Authenticator will take over the validation process and trigger the asynchronized validation callbacks (step 6).

8. DISCUSSION

It is worthwhile to note that the a validation configuration actually provides sufficient information of a particular trust model. Assume that trust anchors are expressed in raw mode in a configuration, any device with such configuration would be able to validate the data

```
policy {
 rule {
  filter {
   packet-type data
   packet-name <>*<KEY><><ID-CERT><>
  }
  checker {
   signature-type rsa-sha256
   min-key-size 2048
   key-locator{
    name <CA><>*<KEY><><ID-CERT>
   }
  }
 }
 rule {
  filter {
   packet-type data
   packet-name <>*
  }
  checker {
   signature-type rsa-sha256
   min-key-size 2048
   key-locator {
    k-pattern (<>*)<KEY><><ID-CERT> \1
    h-relation is-prefix-of
    p-pattern (<>*) \1
   }
  }
 }
}
trust-anchor {
 dir /usr/local/root-ca/keys
}
data-buffer {
 cache-size 1000
 cache-algorithm fifo
 max-freshness 1m
}
```

Figure 14: An example of TLS-style validation configuration.

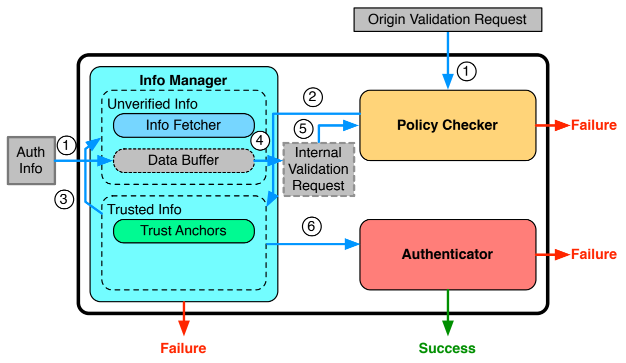

Figure 15: The validation process in TLS mode.

specified in the policy. When a data consumer sends an interest along with the corresponding validation configuration, the intermediate devices (such as routers) will be enabled to validate packets, thus further preventing cache poisoning attacks.

9. CONCLUSION

In this paper, we systematically analyzed the packet validation procedure in NDN application, and identified the packet validation requirements of NDN applications. Based on our analysis, we proposed a modularized validation framework: Athena. The modularized design makes Athena easy to extend.

We also defined a configuration based interface to customize Athena and also defined a policy language Guardian for policy specification. The configuration interface and the policy language free application developers and researchers from implemention details of the validation process, allow them to focus on experimenting different trust model and other security features.

10. REFERENCES

[1] N. project team. NDN packet format specification. http://named-data.net/doc/ndn-tlv/index.html.