Signing of Reference Nets*

Marcel Hansson[1], Marvin Brendel[1], Daniel Moldt[1] and Laif-Oke Clasen[1]

[1]University of Hamburg, Faculty of Mathematics, Informatics and Natural Sciences, Department of Informatics,
http://www.paose.de

Abstract

Construction of complex models, such as colored Petri nets or reference nets, is typically undertaken by distributed teams. To be able to work collaboratively on models, it becomes necessary to exchange models in some way. In some cases, a local execution of exchanged models may also be necessary.

Given the Java inscriptions of reference nets, verifying the absence of harmful code embedded within the net is a complex task. The exchange of such nets and local execution without some check can pose a significant risk. The central question examined in this study is the following: How to ensure the secure import and export of these nets from other modelers?

In the event that a modeler trusts another person and can check their cryptographic signature, it is sufficient if the nets are signed. We developed a prototype for the Petri net editor and simulator Renew to evaluate the proposed concept for reference nets specifically.

The encryption keys utilized for signing the reference net can be obtained through conventional means. A prototype of the NetSigner plugin has been integrated into the tool environment Renew to sign reference nets with trusted keys and to verify the signature of signed reference nets. The subsequent discussion will address the conceptual and technical aspects of signing and verifying net signatures to ensure secure exchanges.

Our working prototype shows that it is possible to ensure a secure import of nets based on signatures. The exchange of complex reference nets, which are not easily verified for the absence of harmful code, is now directly supported in the tool Renew.

Keywords

Authenticity, Certificates, Integrity, Petri Nets, Reference Nets, Renew, Signing, Signatures, Validation

1. Introduction

The development of complex system models, such as reference nets, is increasingly carried out in distributed and collaborative environments. In such settings, models are exchanged between the different participants and in certain cases also locally executed. This introduces significant security challenges. In particular, reference nets may contain embedded executable Java code, making it difficult to assess their safety through manual inspection alone. As a result, importing and executing externally obtained models without proper verification can pose substantial risks.

We want to answer the question: How to ensure the secure import and export of nets from other modelers? A central requirement in this context is the ability to ensure both the authenticity and integrity of exchanged models. Recipients must be able to verify that a given net originates from a trusted source and that it has not been altered during transmission. Traditional approaches to code verification are often insufficient or impractical for complex reference net-based systems.

In this paper, we address these challenges by proposing the use of *cryptographic signatures* for reference nets. These are already commonly used for dealing with these challenges in the general case. By leveraging these established concepts in the context of reference nets, *signed reference nets* enable recipients to validate the origin and integrity of a model consisting of reference nets prior to its execution. To demonstrate the feasibility of this approach, we created a tool called the NetSigner plugin that uses signatures on reference nets within the software Renew. We will present the design, implementation and example usage of this tool in this paper. In addition, we will present a few Use Cases, that motivate the development of the plugin.

PNSE'26, International Workshop on Petri Nets and Software Engineering, 2026

*Supported by participants of our teaching project classes and student theses.

The remainder of the paper is structured as follows: We describe the foundations for our work (Section 2) and describe the set objectives for this paper (Section 3). We then describe our prototype, the NetSigner plugin (Section 4) and present an example usage (Section 5). Afterward we discuss a few use cases that we envision for the NetSigner plugin (Section 6). We examine related work (Section 7) and discuss our plugin (Section 8). Finally, we provide a summary and present possibilities for future work (Section 9).

2. Foundations

2.1. Renew

The Reference Net Workshop Renew [1] is a Petri net editor and simulator originally developed by Olaf Kummer. Its primary focus are the modeling and simulating of reference nets, which will be briefly described in the next section. Since its release it has been continuously expanded with various plugins, allowing for multiple different formalisms and modeling techniques or expanding Renew with other features. With all these expansions, Renew can be considered an integrated development environment for Petri net-based applications. [2]

2.2. Reference Nets

Reference nets [3] are a form of high level Petri nets. Their main feature are other reference net instances as tokens to allow for a nets-within-nets mechanism [4]. The various reference net instances communicate via synchronous channels. Additionally, they allow Java inscriptions on transitions and Java objects as tokens. These inscriptions allow calling any arbitrary Java method, including user-defined methods in custom classes.

Of particular note for a later mention are so-called *shadow nets*. These are an abstraction layer between reference nets including their graphical display (stored as `rnw` files) and the internal simulated reference nets. *Shadow nets* abstract from the graphical representation by excluding all information (e.g. layout, coloring), which does not carry semantic meanings. In addition, multiple *shadow nets* can form one *shadow net system*, which can be stored as one `sns` file. Thus, for simulation purposes, it is enough to have the shadow net files, which, to get to the topic of this paper, are even harder to manually check for malicious code.

2.3. Terminology

This paper uses a number of theoretical terms commonly used in the field of cybersecurity. To improve clarity, the terms used will briefly be explained in this section.

The term *authenticity*[5, 6] describes an entity's property of being genuine. This can mean that said entity is verified and can be trusted. In the context of this paper, this can mean that a reference net received from some other party does indeed come from the source that it claims to come from. The word *authentic* in turn describes that something has verified authenticity.

The term (data) *integrity*[5, 6] refers to the property that something has not been modified in an unauthorized manner. In this paper's context, this can mean that a reference net that was received from some other party has not been altered by a third party during transport.

2.4. Signatures

When receiving data from another, without further measures, one cannot generally be certain of the data's authenticity. Furthermore, even if the data's authenticity can be ascertained, the data's integrity may still yet be dubious. These properties are important though when receiving reference nets and subsequently executing them. Since reference nets (See Section 2.2) can contain executable code, blindly executing one without confirming both its authenticity and integrity might be potentially dangerous.

Signatures provide a way of guaranteeing both authenticity and integrity of received data. [7, 8] The general procedure for signatures is as follows.

Each party has an associated *key pair*. That key pair consists of one *private key* that only the party itself knows and is secret to everybody else. The key pair also contains a *public key* associated with the private key. This public key gets shared with everybody, meaning it is known to all other parties. Any two parties that want to share data use an agreed on *encryption algorithm*. Common examples from reality for such encryption algorithms are *RSA*[9] and *ECDSA*[10]. The signer of a piece of data uses this cryptographic function in combination with their private key to create the *signature* for the data. Then, the signer sends the data alongside with the signature to the recipient of the data. Once the recipient has received the data and signature from the signer, it uses the agreed on encryption algorithm in combination with the signer's already known public key to verify the signature. If that fails, the recipient knows that the data cannot come from the signer. Only the signer's private key should be able to encrypt the data in such a way that their already known public key can verify that.[11]

It is important to note that signing some data is *not* the same as encrypting data. Without encryption and just signing, the data is still unaltered and visible to everybody that can read the channel over which the data is transferred.

2.5. Certificates

Signatures allow a signer to send data that verifiably originates from them. They do not, however, assure that the signer themselves is trustworthy. An actor can use signatures to prove that they are indeed the author of some data. But the recipient of that data has, without further actions, no way of knowing if the signer themselves can be trusted. This problem is being solved by *certificates*. These allow a signer to prove that they are trustworthy to a recipient.[12, 7, 8]

A certificate consists for one part of information about the certificate's subject e.g. country code, organization name, email address etc. In addition, a certificate contains the public key of the certificate's subject. It may also contain a number of extensions and a validity period after which the certificate cannot be used anymore and has to be reissued. Finally, a certificate also contains information about an issuer, which is the base of that certificate's validity. This means, another party guarantees the trustworthiness of the certificate. This other party and certificate issuer is also called *trust authority*. They are the ones who create the certificates on the basis of the data that the certificate's owner provided them with. To prove the credibility of the certificate, the trust authority adds their own signature to the certificate. [7, 8]

This means in effect that the credibility of a certificate is based on another party. This in turn raises the question of what makes such a trust authority trustworthy. Usually, the trust authority has its own certificate proving their own trustworthiness, issued by yet another trust authority. This step may repeat multiple times, creating so-called *certificate chains* or *chain of trust*. [8, 7, 13] This approach still leaves open the base of trust though. In practice, there are multiple standards being used, following different strategies to be able to guarantee this.

The *X.509* standard[13] is widely used to provide a base of trust using a centralized approach. Among a range of areas, it is being used for HTTPS for example. This standard requires a set of entities that serve as *trust anchors*, meaning all credibility for any certificate leads back to one of these. [13, 12] These trust anchors can then just use self-signed certificates. The X.509 does not specify a single set of trust anchors that is generally trusted by everyone [13], but Java does for example already provide a set of trust anchors[1][14].

Alternative standards to X.509 like *PGP*[8] don't build on centralized trust anchors but instead build on a so-called *Web of Trust*. This means several parties verify their trust to each other. PGP is being used for signing emails for example. [8, 7]

[1]The command `keytool -list -keystore $JAVA_HOME/lib/security/cacerts` can be used for inspecting these preset default trust anchors.

2.6. Key Stores

Key stores or *trust stores* are secure collections which may be protected by a password. For example trust store implementations like *Java KeyStore (JKS)* and *PKCS#12* store their contents in a single file. They are supposed to securely store things like private keys, certificates and certificate chains. [15, 16, 17, 18, 19]

3. Objectives

The question that this paper wants to answer was stated in section 1. Namely, the question being asked is how to ensure that shared executable reference nets are authentic and have not been tampered with by third parties. These problems are already commonly being solved for shared data in general with the approach of using signatures (See Section 2.4) and certificates (See Section 2.5). Therefore, it is reasonable to apply this approach to executable nets, such as reference nets, in an attempt to solve the question.

As of yet, no such application of signatures and certificates to guarantee authenticity and integrity (See Section 2.3) exists for the distribution of executable nets in general (See Section 7). Therefore, in order to answer the question posed, the *first objective* of this paper is to develop a solution that applies the employment of signatures and certificates to the transfer of executable nets, specifically reference nets in this case. Part of the objective should be to describe this solution within this paper.

The solution should be developed within the context of the reference net editor Renew (See Section 2.1). The reason being that Renew provides an environment for creating and editing, as well as executing reference nets. Furthermore, Renew's plugin system allows for the optional addition of further types of nets. Within the scope of this paper, the goal is only to prove the feasibility of applying signatures and certificates to executable nets. This paper is therefore limiting itself to applying these concepts to reference nets within Renew. Expanding this application to other types of nets should however well be possible for future research within the context of Renew.

While only describing the solution developed for this paper would give readers insight into the extent of its capabilities, it would still lack reproducibility. Therefore, the *second objective* of this paper is to showcase the solution found for the first objective. This should happen by leading the reader through practical examples on how features of our solution can be used in practice.

In order to demonstrate the usefulness of our solution, the *third objective* is to show how our solution could be employed in the real world. For that, a set of possible practical use cases should be named where our solution could be applied to. Alternatively, this may also include fields where the general exchange of executable nets with guaranteed authenticity and integrity would be beneficial.

4. The NetSigner Plugin

The problem that is being addressed by this paper is the question whether a reference net is trustworthy when it is received from some other party. Since reference nets may contain executable Java code, blindly executing such a net after receiving it from another party may be potentially dangerous. This problem can generally be solved by using certificates (See Section 2.5) which in turn utilize signatures (See Section 2.4). These ensure that data verifiably comes from the source it claims to come from and that the data has not been modified by third parties during transfer.

However, previously, there existed no such implementation of certificates for executable nets, including reference nets (See Section 2.2). This paper introduces such an implementation in the form of a prototype for reference nets that are verified by signatures. In particular, a new plugin for the net editor Renew (See Section 2.1) was developed. This new plugin, one of many optional plugins for Renew, was originally created by [20] and was dubbed the NetSigner plugin.

This plugin in particular extends Renew's functionality by introducing *signed nets*. For now, due to technical reasons, these nets can only be of the type reference nets. With the plugin, a reference

net can be exported as a signed reference net. To be exact, Renew can save reference nets as files by serializing these to a string of data, which is being saved in a file. When a reference net is exported as a signed reference net, that string of data is saved as a file as well, but with the addition of a signature for that string of data as well as the signer's certificate. These signed reference nets are stored as a **srn** file. This signature allows any other user of Renew to verify that a signed reference net file is authentic. In addition, users can validate that a reference net has not been altered by third parties at any point because it was signed. Optionally, reference nets may also be exported as signed shadow net systems instead (See Section 2.2). In that case the signed net is stored as a **ssn** file.

Reference nets can also be imported into Renew from these signed reference net files, where the signature of the signed reference net will be verified first. Subsequently, the trustworthiness of the net's author themselves is verified using certificates. Only then will the reference net be read from its binary data and loaded into the Renew runtime. There, it can be viewed, executed and modified like any other reference net in Renew.

In addition to the import and export of reference nets as signed reference nets, the NetSigner plugin offers users a number of convenience features. This includes the ability to directly inspect the author of any loaded signed reference net and list all loaded signed nets. Furthermore, users may directly verify the signature of a signed reference net file and read its author without having to directly open the net in Renew.

By default, the NetSigner plugin handles all signatures via an implementation using the *X.509* standard (See Section 2.5). This implementation supports multiple encryption algorithms, e.g. *RSA*[9] or *ECDSA*[10]. The plugin also provides an interface for implementing and using other certification standards so that other standards may be added by anyone. In addition, the plugin is configurable so that the default X.509 implementation can be deactivated[2].

In order to use the X.509 implementation to sign a reference net, a user needs a certificate that is issued by a source that is trusted by the recipient of the signed reference net. This source of trust will usually be some entity that is either an official trust anchor[3] or some entity that can trace its credibility back to an official trust anchor. To facilitate acquiring an official certificate, the NetSigner plugin provides an interface for creating a certificate signing request (CSR). This produces a CSR file that may be used when requesting a certificate at an official site. Alternatively, the NetSigner plugin also offers users the ability to specify custom trust anchors. This can be done by providing a trust store (See Section 2.6) containing those custom trust anchors' certificates[4]. This way, the password does not have to be entered manually after each time Renew starts.

To sign a reference net, the users need to provide a valid certificate proving their identity, as well as the private key associated with that certificate. Both are stored in a single JKS key store file. The key store's location as well as the key's name need to be specified by the user[5]. Users can also just import the certificate from a file by using the Renew interface. The file formats **crt** and **pem**, as well as **p7b** and **p7s** are valid file types for this action. Alternatively, a *PKCS#12* key store may be imported from a **p12** file, which can contain a certificate and the private key associated with it. The key store and the private key it contains may be protected by a password. Either the Renew interface will prompt the user for those passwords once they are needed, or they can be specified by editing a configuration file[6] to skip this user interaction. In the former case, the Renew runtime will remember those passwords so that they only need to be supplied once by the user per session.

In summary, the NetSigner plugin, in its core, allows users of Renew to share reference nets with each other securely by ensuring the authenticity as well as the integrity (Section 2.3) of the transferred net data. This trust between sender and receiver is achieved by the two core functionalities provided by the plugin. Firstly, the added ability to attach signatures to these reference nets, i.e. *signing these*

[2]The property `de.renew.netsigner.useX509Signer` needs to be set to `false`. See [1]

[3]As specified by the default trust anchors provided by Java.

[4]The trust store file's location must be specified by the property `de.renew.netsigner.trustAnchors`. Optionally, the trust store's password can directly be configured as well with `de.renew.netsigner.trustAnchorPassword`.

[5]By using the properties `de.renew.netsigner.keyStorePath` and `de.renew.netsigner.keyName`.

[6]By using the properties `de.renew.netsigner.keyStorePassword` and `de.renew.netsigner.keyPassword`.

nets. And secondly, a way for the receiver to validate the signature by checking it against the received reference net data.

5. Examples

In order to facilitate understanding how to use the NetSigner plugin, the basic workflow of signing a reference net (Section 5.1) as well as reading a signed reference net (Section 5.2) will be illustrated here.

5.1. Signing a reference net

Imagine you just created a reference net in Renew and you want to share the net with somebody else in another institution. If they would just receive the raw net out of the blue, they might not know that it was in fact you who sent them that reference net. Furthermore, neither you nor the receiver of the net can be sure that nobody intercepted the net, modified it and then sent it to the receiver. Since the reference net can execute code on the system, this can pose a potential threat.

Therefore, you decide to not just send them the raw net, but instead you send them a *signed reference net* by using the NetSigner plugin for Renew. In order to sign a reference net with the NetSigner plugin, the first thing that is needed is a certificate, as well as the associated private key. Normally you would request a certificate from some official certificate authority which would be trusted by the recipient of your net. They should then give you a certificate file[7].

Obtaining that certificate might be a bit lengthy of a process though, and can differ depending on the certificate authority. For demonstration purposes, a self-signed certificate will be used for this example. Both kinds of certificates, one issued from an official authority as well as a self-signed certificate, work the same though. What matters is that the trust anchor of the certificate is trusted by the recipient. A quick self-signed certificate can be generated by the following command.

For this example, one key alongside a self-signed certificate will be generated with this command:

```
^^Iopenssl req -x509 -newkey rsa:2048 -nodes \
^^I^^I-keyout mykey.pem -out mycert.pem \
^^I^^I-addext "keyUsage=digitalSignature,keyEncipherment"
```

The command **`openssl req`** is the default command for creating a self-signed certificate on Linux systems[8]. Usually it is used to create certificate signing request file that can be submitted at a certificate authority to request a certificate. The **`-x509`** specifies that the command should produce a self-signed certificate though. The argument **`-newkey rsa:2048`** specifies that a new key pair will directly be generated for the certificate. And it is using the RSA cryptographic algorithm and a key size of 2048 bits. Using **`-nodes`** makes the key unencrypted, meaning it will not be protected by password. Since this is only a quick showcase example to obtain a self-signed certificate this is alright. In practice, certificates should be obtained from a trust authority and protected by a password. The argument **`-keyout mykey.pem`** writes that key to a file called `mykey.pem`. The certificate itself is written to a file called `mycert.pem` as specified by the **`-out mycert.pem`** argument. In order to be usable for signatures, the argument **`-addext "keyUsage=digitalSignature,keyEncipherment"`** is necessary. This specifies that the key uses the extensions `digitalSignature` and `keyEncipherment` which are necessary for the certificate to be able to create signatures.

Once you have a certificate, you can load it into Renew in order to sign reference nets. One way to do this is to start Renew and then load the private key for the certificate as well as the certificate itself independently. Alternatively, in order to facilitate the import of private key and certificate into Renew, a PKCS#12 key store can be used as well. The advantage would be that the store contains both key and certificate, so that only one file needs to be imported. Based on the previous command that generated the files `mykey.pem` and `mycert.pem`, the following command can be used for this.

[7] Usually in the file format `crt` or `pem`

[8] Other types of OS might use different commands

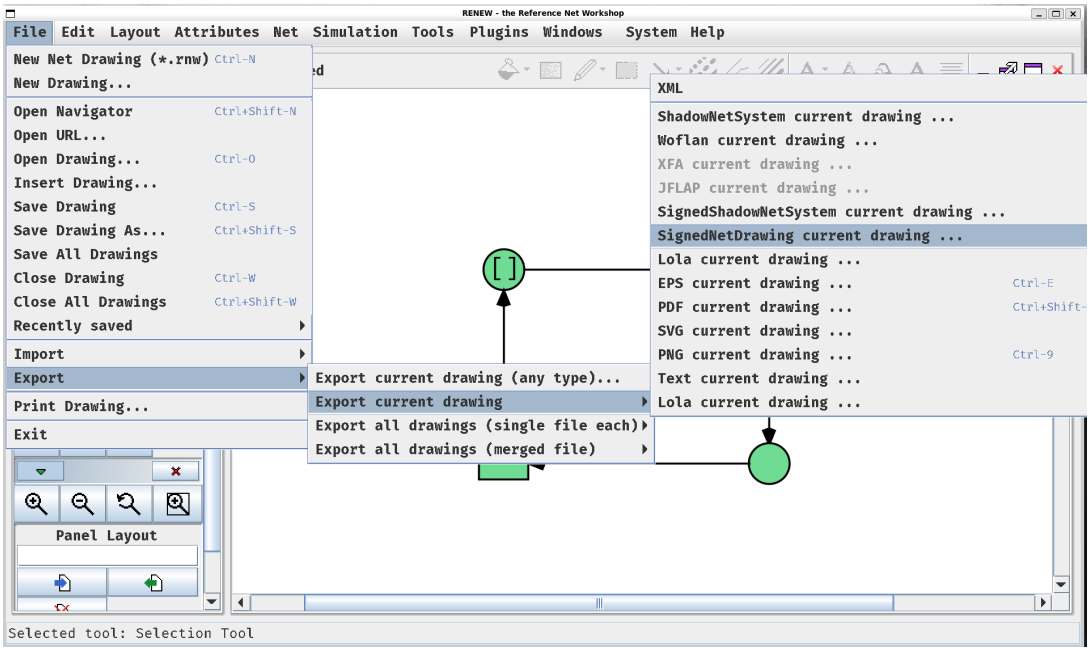

Figure 1: Export a reference net as a signed reference net

```
^^Iopenssl pkcs12 -export \
^^I^^I^^I-inkey mykey.pem \
^^I^^I^^I-in mycert.pem \
^^I^^I^^I-out mycert.p12
```

This command uses the executable **`openssl pkcs12`** which is the default for Linux systems. Other sorts of systems might use other commands for creating a PKCS#12 store. The argument **`-export`** specifies that a PKCS# store should be created, instead of reading or extracting data from an existing store of that type. The part **`-inkey mykey.pem`** specifies the key to be used for that store, namely the previously created `mykey.pem` file. Similarly, the argument **`-in mycert.pem`** specifies the input certificate. The argument **`-out mycert.p12`** specifies the file to which the produced PKCS#12 store should be written. Thus, this command produces the file `mycert.p12` which contains the PKCS#12 store holding both the private key and the certificate.

After the certificate and the associated private key are obtained, RENEW can be started. That requires setting some properties for the NETSIGNER plugin beforehand though. This can be done by creating a file called `renew.properties` and moving it to the location specified by [1]. For this example, these properties should be set like this in that file:

```
^^Ide.renew.netsigner.keyStorePath = config/signing.jks
^^Ide.renew.netsigner.keyName = mycert
```

The property `de.renew.netsigner.keyStorePath` defines the location of the key store that will hold both the private key for a certificate and the certificate itself. In this example, it is set to `config/signing.jks`. That means that the file will be stored in a directory called `config` relative to the RENEW executable. The filename itself of that key store can be chosen freely, it just needs to have the file type `jks`. This file does not need to be created before starting RENEW for the first time, as it will automatically be created if it does not already exist. If it does already exist however, RENEW is going to reuse this file. In turn, this means that both private key and certificate only need to be imported into RENEW once per machine and can then be accessed again at any time. The property `de.renew.netsigner.keyName` defines the name of the private key used for the certificate. The name can be chosen freely and may even be omitted.

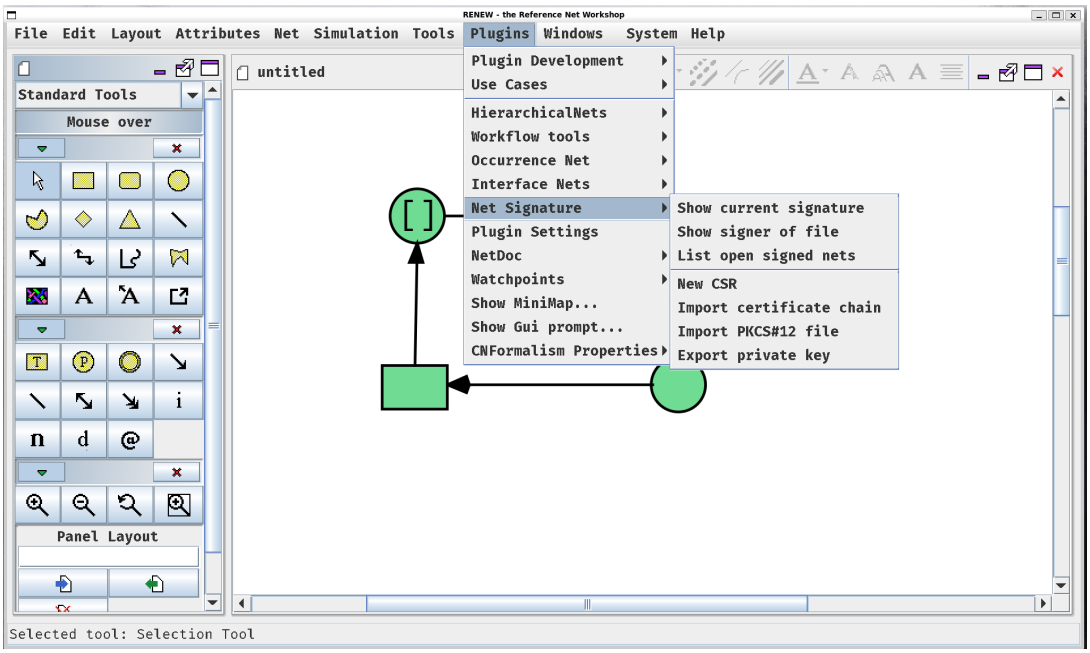

Figure 2: Actions for the NetSigner plugin

Once these properties have been configured, Renew can be started. In order to do that, the command java -p path/to/renew:path/to/renew/libs -m path/to/renew/de.renew.loader gui is used[9]. Once Renew completed startup, the private key and the self-signed certificate that were just created must be loaded into Renew. These could be imported independently, but the more convenient way is to directly import the PKCS#12 file that was just created for this example. See Figure 2 on how to do that. This will load both items into the JKS key store used by the NetSigner plugin. Since these items are both stored in one file, the action of importing the private key and the certificate chain only needs to be done one time. After that, Renew can be stopped and started again, but this action won't be necessary anymore.

Once private key and certificate are loaded into Renew, create a reference net or open a previously created reference net[10]. Afterwards, the net can be exported as a signed reference net (see Figure 1). This will create a **srn** file which contains the reference net itself alongside a signature for that net. That signature is created using the private key and the certificate which is loaded in the JKS key store that is used by the NetSigner plugin. Alternatively, the net can also be exported as a signed *shadow net system*. This means all the information of the net is preserved like what components it consists of and how these are connected to each other, but all graphical information about the net is not preserved. In this case, the produced file will have the **ssn** file type.

5.2. Reading a signed reference net

This example shows the process of loading a signed reference net into Renew. In this case, the signed reference net that was created as part of the previous example will be loaded into Renew.

Reading a signed reference net means that you want to import either a **srn** or **ssn** file into Renew and check both its authenticity and integrity (See Section 2.3). Since the NetSigner plugin uses the X.509 certificate standard by default, Renew will check if the author of the signed reference net is trustworthy. This means that the author of any signed reference net must be able to trace their own credibility back to a trust anchor that is trusted by the recipient of the net. By default, this is going to be one of the default trust anchors specified by Java. The NetSigner plugin allows specifying custom trust anchors as well. For importing the signed reference net from the last example, it is necessary to

[9]On Windows, use java -p "path\to\renew;path\to\renew\libs" -m path/to/renew/de.renew.loader gui instead.
[10]See [1] for how to do that.

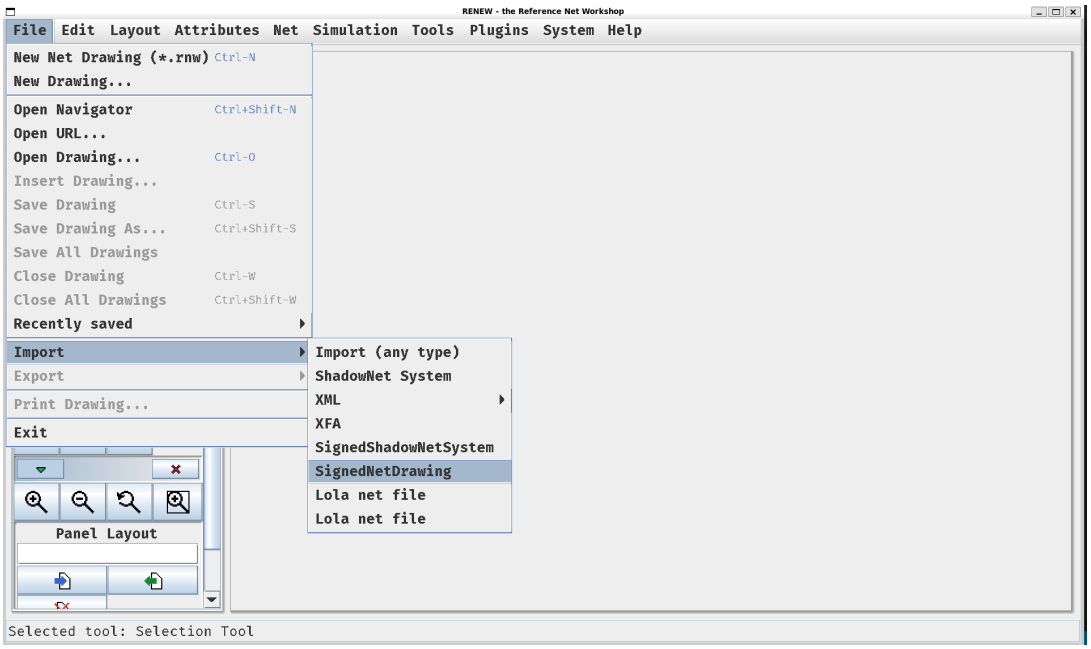

Figure 3: Import a signed reference net

specify the self-signed certificate that was created there as a custom trust anchor. In order to do that, a trust store must be created first that contains the self-signed certificate. This is what the following command does:

```
^^Ikeytool -importcert \
^^I^^I-alias mycert \
^^I^^I-file mycert.pem \
^^I^^I-keystore trust.jks \
^^I^^I-storepass <custom password>
```

The command **`keytool`** is the default command for interacting with JKS key stores. By default, it is already installed when Java is installed on a system. The argument **`-importcert`** signifies that the keytool command should import a certificate. Next, the argument **`-alias mycert`** gives the certificate being imported an identifier. In this case the identifier *mycert* is chosen. The argument **`-file mycert.pem`** specifies the input file containing the certificate. The argument **`-keystore trust.jks`** specifies the target trust store file that this command will produce. The name `trust.jks` was chosen for this example, however it may be chosen freely. It will just need to be referenced later in Renew. Finally, trust stores require a password. This password will be set by using the argument **`-storepass <custom password>`**. The password should be chosen by the user themselves.

Once the trust store is created, it still needs to be referenced in Renew. In order to do that, these entries must be added to the `renew.properties` file used by Renew (See Section 5.1):

```
^^Ide.renew.netsigner.trustAnchors = path/to/trust.jks
^^Ide.renew.netsigner.trustAnchorPassword = <custom password>
```

The property `de.renew.netsigner.trustAnchors` needs to be set to the path of the trust store file that was just created for this example. The password for that trust store can directly be set with the property `de.renew.netsigner.trustAnchorPassword`. This needs to be the same password that was provided by the last command that created the trust store.

With these properties set, Renew can be started again. Once startup has completed, the signed reference net can be imported (see Figure 3). This will load the signed reference net into Renew. Before

the potentially untrusted reference net itself gets loaded though, the NetSigner plugin will first verify the signature of the net. In this case, the author of the signed reference net used a self-signed certificate which was loaded into the trust store of custom trust anchors for the NetSigner plugin. Therefore, this signed reference net has a trusted author.

Would the signed reference net have an author that can't trace their trustworthiness back to any accepted trust anchor, the reference net would not get loaded. Thus, the import would fail with an error message signaling that the author of the signed net is not trusted. Likewise, had the signed reference net been modified in any way by a third party, the NetSigner plugin would not load the signed reference net either, signaled by a custom error message.

However, since none of these circumstances apply for this example, the signed reference net gets loaded into Renew. There, the net can be inspected as well as modified and may be executed.

6. Use Cases

The concepts introduced in this paper are illustrated by four closely related application scenarios: distributed simulation, distributed verification, collaborative modeling, and the public dissemination of Petri net artifacts. Across all four scenarios, the contribution is situated at the artifact level. Its purpose is to enable cryptographically verifiable provenance, integrity, and release attribution. This applies to exchanged Petri net artifacts. Importantly, this does not extend to establishing semantic correctness, behavioral adequacy, or analytical validity of the model itself.

This distinction between artifact-level guarantees and semantic correctness is essential to understanding the boundaries of the contribution. A digital signature can authenticate a concrete artifact version and protect it from undetected modification after release. However, it does not imply properties such as soundness, boundedness, liveness, or suitability for a given task. The contribution, therefore, addresses trust and traceability in Petri net workflows rather than formal correctness. Although the implementation considered here uses reference nets as examples, the underlying artifact-level concepts are transferable to other Petri net classes. This transfer is possible if their representations and signable artifact units are precisely defined.

In distributed simulation, multiple Petri net modules may be composed and executed across separate simulation instances. [21, 22, 23, 24] In this setting, signatures can be used to verify that exchanged model artifacts originate from a trusted issuing entity and have not been altered prior to execution. The NetSigner plugin provides an example of such an implementation. These guarantees concern only the authenticity and integrity of the artifacts. They do not and cannot certify the technical or semantic correctness of the artifacts being exchanged or analyzed.

A second application scenario is distributed verification. [25, 26, 27, 28, 29] Here, models, submodels, property specifications, or derived analysis artifacts may be exchanged across tools or organizational boundaries. For a scientifically meaningful interpretation of verification results, the underlying artifacts must be identifiable in a stable, integrity-protected form. Digital signatures can support this. They bind released artifacts to a specific issuing entity and enable later verification that the analyzed version has remained unchanged. They thus improve traceability and auditability. However, full reproducibility also depends on semantics, tool behavior, parameters, and metadata.

A third scenario is collaborative modeling. [30, 1, 31] When multiple actors jointly develop Petri net models, signatures can be used to designate approved model states and to verify their provenance and integrity across subsequent editing and exchange steps. This is particularly relevant in distributed development and heterogeneous toolchains, where accountable release states are needed.

A fourth scenario is the public dissemination of Petri nets through repositories and comparable infrastructures. In this context, signatures support verifiable attribution of published artifacts to a specific source. They also protect against unnoticed post-publication modification. This strengthens traceability and supports reliable referencing and reuse of published models. Full scientific reproducibility, however, still requires additional contextual information. Relevant infrastructures include Petriweb [32] and the Petri Nets Repository [33].

Taken together, these scenarios show that digital signatures can serve as a focused infrastructure component for provenance-aware handling of Petri net artifacts across simulation, verification, collaboration, and publication. Their contribution is scientifically relevant but circumscribed: they enable verifiable artifact attribution and integrity checks across trust boundaries. However, they do not replace or obviate semantic analysis or correctness arguments for the artifacts themselves.

7. Related Work

There exist various approaches for the secure exchange of software artifacts, addressing concerns such as authenticity, integrity, and trustworthiness of distributed code.

A well-established technique in this domain is code signing, which ensures both the authenticity and integrity of software artifacts through cryptographic means. Widely used mechanisms, such as Java Code Signing [34] and Microsoft Authenticode [35], rely on public key infrastructures (PKI) to verify the origin and integrity of executable code prior to execution. These approaches have become standard in modern software distribution and are deeply integrated into development and deployment workflows.

An alternative line of work is proof-carrying code [36], where safety is guaranteed by attaching formal proofs to the code itself. In contrast to signature-based approaches, proof-carrying code does not rely on external authentication or cryptographic infrastructures, but instead enables the receiving system to independently verify compliance with specified safety properties.

Closely related is the concept of software provenance, which addresses the traceability of software artifacts and their origins [37]. Provenance research emphasizes the importance of metadata, including version histories, authorship information, and transformation records. Practical guidelines, such as those provided by industry sources (e.g., JFrog [38]), highlight cryptographic signatures as a key component for ensuring trustworthy provenance in software supply chains. However, these approaches are generally designed for traditional software artifacts and do not explicitly consider executable modeling formalisms.

More recently, similar challenges have emerged in the context of machine learning and artificial intelligence. The increasing distribution and reuse of trained models has raised concerns about integrity and security, particularly as models may embed executable components or trigger arbitrary code execution when loaded. Maliciously modified models distributed through compromised repositories pose a significant risk. Consequently, signing and verification mechanisms for AI models have been proposed as a means to ensure their integrity and trustworthiness [39]. This development highlights comparable integrity concerns for non-traditional executable artifacts.

However, these approaches are primarily designed for conventional software artifacts and do not directly transfer to executable modeling languages such as reference nets. The combination of graphical structure, embedded code, and tool-specific representations introduces challenges that are not addressed by existing signing and provenance mechanisms. In particular, there is little work on integrating authenticity and integrity verification directly into modeling environments. This gap motivates the approach presented in this paper.

8. Discussion

In this section, the contributions from the previous parts of this paper are going to be evaluated on the basis of this paper's research question and objectives (See Section 3). First, we will answer the research question we posed in this paper in section 8.1. Afterwards, we will evaluate to what degree we met the objectives we set for ourselves in section 8.2. Lastly, in section 8.3, we will discuss the advantages and disadvantages as well as the current limits of our developed solution.

8.1. Research Question

The question this paper asks is how to share executable reference nets in a distributed environment while guaranteeing authenticity and integrity (See Section 2.3). Our solution to that question was to *sign* these nets, i.e. utilizing *signatures* and *certificates* to guarantee these desired properties. While signatures and certificates are already commonly used for data in general, there existed no specific application of these tools for executable nets specifically yet (See Section 7). Our contribution was to create an implementation of that concept within the reference net editor and simulator Renew. Thereby, we proved the feasibility of applying signatures and certificates on shared executable nets in order to guarantee authenticity and integrity.

8.2. Objectives

We were successfully able to achieve the objectives we set for ourselves (See Section 3) in this paper. For the first objective, the plugin NetSigner was added to Renew. This plugin adds the core functionality of exporting reference nets as signed reference net files, which requires the user to have a valid certificate to prove their credibility. This signed reference net file can then be imported again, whereupon the signature of the signed reference net file is verified. Should the signature check fail, the net is not imported, so it cannot be executed locally either. This solution was described in detail in section 4, thereby fulfilling our first objective.

For our second objective, we wanted to improve the reproducibility of the solution we developed. We fulfilled this objective by guiding the reader through practical examples on how to use core functionalities of the NetSigner plugin (See Section 5).

Lastly, our third objective was to show the scope of what our solution, as well as using signatures and certificates on executable nets in general, could be used for. This was done in section 6, thereby fulfilling the third objective we set for ourselves.

8.3. Evaluation of the NetSigner Plugin

Our solution successfully manages to guarantee authenticity and integrity for shared reference nets by employing the common approach of using signatures and certificates. This was achieved with the development of the *NetSigner* plugin for the Petri net editor Renew. With the use of this plugin, reference nets can be exported as *signed reference nets* files. This means that the binary representation of a reference net is coupled with its signature. Given that only the signer themselves possess the private key needed to produce a signature, no third parties are able to forge these signatures. All the while, everybody possessing the corresponding public key can verify those signatures.

In addition, the author of the net has to be verified via a certificate. This means that no third parties can distribute nets with malicious intent.

These signed reference net files can be imported with the NetSigner plugin. The plugin automatically verifies the signature along with the associated certificate. This mechanism guarantees the properties of authenticity and integrity for any reference net received in this manner.

The default implementation of this plugin uses the X.509 standard (See 2.5) which uses certificate chains leading back to a trust anchor. While Java supplies a set of default trust anchors, the NetSigner plugin also offers users the ability to use a set of custom trust anchors. I.e. users are given the liberty to choose which parties they want to trust, and are not forced to work with parties against their will for the sake of securely sharing executable nets.

Another advantage of the NetSigner plugin is the uncomplicated handling of certificates. While the acquisition of an official certificate itself can be tedious, once it is obtained, it can be integrated relatively easily into the NetSigner plugin. The certificate just needs to be imported once into Renew alongside the related private key and then the plugin automatically uses the certificate and key for signing any reference nets. Even when Renew is shut down and started again, the certificate and private key persist.

In addition, the NetSigner plugin offers some convenience features for its users. For example, the certificates of signed nets can be inspected in the GUI, signatures of signed nets can be verified without loading the net. Furthermore, the plugin provides a form for generating a certificate signing request, which might be used for the acquisition of a certificate from some authority.

A disadvantage of our solution is that the involvement of signatures and certificates creates a certain overhead. The secure exchange of reference nets cannot just be used ad hoc with our solution but instead requires some previous setup first, like importing a valid certificate and key. In conjunction with that, the acquisition of a valid certificate can be tedious. It usually involves an official request at some authority which can also differ depending on the exact certificate issuer. Even if a custom trust anchor inside the own organization is being used, obtaining a usable certificate will require some form of process on an organization level. It should be mentioned though that this sort of organizational overhead is inherent with the use of certificates. In order to avoid it, some other strategy guaranteeing authenticity and integrity would need to be found.

Currently, the solution we developed is still limited by some aspects, but it could still well be expanded on in the future. For once, at the moment, the NetSigner plugin is only able to sign reference nets. However, all the types of nets supported by Renew and its plugins can be stored as binary data inside files. Therefore, creating signatures for these nets should pose no significant problem. In turn, expanding the functionality of the NetSigner plugin in the future to be able to sign all types of nets supported by Renew should be unproblematic.

Another aspect that could still be improved is the current lack of encryption for shared nets. While signed nets do guarantee authenticity and integrity of a received net, the net would still be readable to any third parties with access to the communication channel. This means that, while a received executable net should always be harmless, it might not be secret from unauthorized third parties. Should the net contain confidential data, this could be problematic. This feature can still be integrated into Renew in the future though and is compatible with signed nets.

Something that still poses a risk with the current implementation of the NetSigner plugin is the following point. Reference nets are a powerful type of net, including executable Java code in inscription and tokens holding whole reference nets. The latter part is still problematic because during simulation, Renew will dynamically load the template of any reference net contained in a token and execute that net. Should this dynamically loaded reference net be some malicious net that somehow made it onto the machine, it would get executed. Therefore, in order to be secure, it is still necessary at the moment to not only check the current net's authenticity and integrity before executing. Instead, all nets that could be dynamically loaded must be manually verified. Future work has to expand our current solution and add some sort of mechanism that only dynamically loads signed nets.

Yet another feature that could still get implemented in the future and is not part of our solution yet is some sort of secure mode for Renew. At the moment, signed nets can only get loaded by importing files holding signed nets. However, normal unsigned nets can still always be opened by Renew. In turn, some reference net containing malicious Java code can still be opened in Renew by the user. In order to avoid this, some sort of restricted secure mode for Renew would be needed where opening or at least executing unsigned nets is not possible anymore.

9. Conclusion

This paper asked itself the question of how colored Petri nets, in particular reference nets (See Section 2.2), can be shared securely among participants in a distributed environment. The problem that was identified with doing that in an entirely unprotected manner is the following. A third party could pretend to be a trusted entity and send reference nets containing harmful Java code to participants in a distributed environment. Likewise, a third party could intercept the transfer of a net, inject malicious Java code or replace the net and then send the harmful net to the original recipient. In summary, authenticity and integrity (See Section 2.3) of transferred reference nets may become compromised without additional measures.

As a solution for this problem, this paper proposes a prototype in the form of the NetSigner plugin for the net editor Renew (See Section 2.1). This prototype employs *signatures* (See Section 2.4) alongside *certificates* (See Section 2.5) in an attempt to solve this problem. Signatures and certificates are already commonly used in order to solve the problem of authenticity and integrity of data in general. However, previously there existed no proper solution for handling the transfer of reference nets regarding their authenticity and integrity. The contribution of this paper lies in the application of signatures and certificates on the new domain of executable nets.

To begin with, we explained some foundations in section 2 that are necessary in order to understand this paper. Namely, this included a short description of Renew (See Section 2.1) and reference nets (See Section 2.2). Subsequently, we clarified the terminology coming from the field of cybersecurity that we used (See Section 2.3). This was followed with an explanation of signatures (See Section 2.4), certificates (See Section 2.5) and key stores and trust stores (See Section 2.6).

After that, we defined the objectives that we set for ourselves for this paper (See Section 3). The first one was to develop an application of signatures and certificates for reference nets and describing the solution. The other two objectives were presenting examples on how to use our solution, and describing possible use cases for it.

For the main part and contribution of this paper, we presented the solution we developed for answering the research question of this paper (See Section 4). Namely, this was the NetSigner plugin which allows users to export any reference net as a *signed reference net* file which a recipient can then import, verifying its signature and certificate. This presentation of the NetSigner plugin was then succeeded by two reproducible examples on how to use the NetSigner plugin's core functionality (See Section 5).

Subsequently, we described a number of possible fields where our prototype and signed nets in general could be employed (See Section 6). Following this, we contextualized this paper within the current field of study by highlighting the related work to this paper (See Section 7).

We then evaluated the paper in section 8. There, answered the research question on the basis of this paper's research (See Section 8.1). Next, we evaluated how well the objectives that were previously set were fulfilled (See Section 8.2). Finally, the current advantages and disadvantages of the prototype we developed were assessed (See Section 8.3). This also included a discussion on the current limits of our prototype.

In the context of further research regarding this paper's topic, one could build on the current prototype and add the ability to sign all types of nets supported by Renew. Since all types of nets are already stored as binary data by Renew, also creating signatures for these nets is basically trivial. This change would require no modification to the developed certificate logic either.

Another aspect that could be combined with signed nets would be to also encrypt them. While signed nets protect against executing potentially harmful code, other security aspects are not addressed by certificates and signatures. Should a shared net contain sensitive information, an attacker could observe the channel over which it is transferred and plainly read the net. Encrypting these nets, in addition to using signatures and certificates would improve security concerns in a distributed environment.

In the current prototype, when a reference net is simulated and holds a reference net in one of its tokens, the type of net gets dynamically determined during runtime, based on the net type's name. This opens the potential for signed reference nets to load an unsigned reference net which itself could contain malicious Java code. As part of future research, security restrictions should be added to Renew that prohibit this. One approach could be to restrict signed nets to only be able to dynamically load other signed nets. Another approach could be to introduce some sort of security mode in Renew where unsigned nets cannot be opened normally or loaded dynamically. Instead, nets could only be loaded into Renew when they are imported from signed nets.

Acknowledgment This research is funded by the Deutsche Forschungsgemeinschaft (DFG – German Research Foundation), grant no. 528713834.

Declaration on Generative AI

During the preparation of this work, the authors used ChatGPT, DeepL, Grammarly and LanguageTool in order to: Grammar and spelling check, Paraphrase and reword. After using these tool(s)/service(s), the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] O. Kummer, F. Wienberg, M. Duvigneau, L. Cabac, M. Haustermann, D. Mosteller, M. Hansson, Renew – the Reference Net Workshop, 2025. URL: http://www.renew.de/, release 4.2.

[2] L. Cabac, M. Haustermann, D. Mosteller, Renew 2.5 - towards a comprehensive integrated development environment for petri net-based applications, in: F. Kordon, D. Moldt (Eds.), Application and Theory of Petri Nets and Concurrency - 37th International Conference, PETRI NETS 2016, Toruń, Poland, June 19-24, 2016. Proceedings, volume 9698 of *Lecture Notes in Computer Science*, Springer-Verlag, 2016, pp. 101–112. URL: http://dx.doi.org/10.1007/978-3-319-39086-4_7.

[3] O. Kummer, Referenznetze, Logos Verlag, Berlin, 2002. URL: http://www.logos-verlag.de/cgi-bin/engbuchmid?isbn=0035&lng=eng&id=.

[4] R. Valk, Petri nets as token objects - an introduction to elementary object nets, in: J. Desel, M. Silva (Eds.), 19th International Conference on Application and Theory of Petri nets, Lisbon, Portugal, number 1420 in Lecture Notes in Computer Science, Springer-Verlag, Berlin Heidelberg New York, 1998, pp. 1–25. URL: https://doi.org/10.1007/3-540-69108-1_1.

[5] R. W. Shirey, Internet Security Glossary, Version 2, Request for Comments RFC 4949, Internet Engineering Task Force, 2007. URL: https://datatracker.ietf.org/doc/rfc4949. doi:10.17487/RFC4949.

[6] H. Kersten, K.-W. Schröder, ISO 27001: 2022/2023: Management der Informationssicherheit nach den aktuellen Standards, Edition <kes>, Springer Fachmedien Wiesbaden, 2023. URL: https://link.springer.com/10.1007/978-3-658-42244-8. doi:10.1007/978-3-658-42244-8.

[7] J. Katz, Y. Lindell, A Graduate Course in Applied Cryptography, Chapman & Hall/CRC Cryptography and Network Security Series, third edition ed., CRC Press, 2021.

[8] C. Paar, J. Pelzl, T. Güneysu, Understanding Cryptography, second edition ed., Springer-Verlag GmbH, 2024. doi:10.1007/978-3-662-69007-9.

[9] K. Moriarty, B. Kaliski, J. Jonsson, A. Rusch, PKCS #1: RSA Cryptography Specifications Version 2.2, Technical Report 8017, Internet Engineering Task Force (IETF), 2016. URL: https://www.rfc-editor.org/info/rfc8017. doi:10.17487/RFC8017.

[10] T. Pornin, Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA), Technical Report 6979, Internet Engineering Task Force (IETF), 2013. URL: https://www.rfc-editor.org/info/rfc6979. doi:10.17487/RFC6979.

[11] A. Menezes, N. P. Smart, Security of signature schemes in a multi-user setting, Des. Codes Cryptogr. 33 (2004) 261–274. URL: https://link.springer.com/10.1023/B:DESI.0000036250.18062.3f.

[12] D. R. Stinson, Cryptography: Theory and Practice, Discrete Mathematics and Its Applications, third edition ed., Chapman & Hall/CRC, 2006.

[13] S. Boeyen, S. Santesson, T. Polk, R. Housley, S. Farrell, D. Cooper, Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile, Request for Comments RFC 5280, Internet Engineering Task Force, 2008. URL: https://datatracker.ietf.org/doc/rfc5280. doi:10.17487/RFC5280.

[14] C. Rosu, Get a List of Trusted Certificates in Java | Baeldung, https://www.baeldung.com/java-list-trusted-certificates, 2020.

[15] Oracle, KeyStore (Java Platform SE 8), 2025. URL: https://docs.oracle.com/javase/8/docs/api/java/security/KeyStore.html#getDefaultType--, accessed: 2026-03-26.

[16] Oracle, KeyStores and TrustStores (Configuring Java CAPS for SSL Support), 2010. URL: https://docs.oracle.com/cd/E19509-01/820-3503/ggffo/index.html, accessed: 2026-03-26.

[17] Oracle, Creating a KeyStore in JKS Format (Configuring Java CAPS for SSL Support), 2010. URL: https://docs.oracle.com/cd/E19509-01/820-3503/ggfen/index.html, accessed: 2026-03-26.
[18] Oracle, Creating a KeyStore in PKCS12 Format (Configuring Java CAPS for SSL Support), 2010. URL: https://docs.oracle.com/cd/E19509-01/820-3503/ggfhb/index.html, accessed: 2026-03-26.
[19] A. Kario, R. H. Inc., Use of Password-Based Message Authentication Code 1 (PBMAC1) in PKCS #12 Syntax, Request for Comments RFC 9879, Internet Engineering Task Force, 2025. URL: https://datatracker.ietf.org/doc/rfc9879. doi:10.17487/RFC9879.
[20] R. Jürgensen, Untersuchung des Zertifikateeinsatzes im Java-Umfeld – Beispielhafte Diskussion von Zertifikaten für die Open-Source Software Renew, Master thesis, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, 2021.
[21] L. Clasen, S. Bartelt, Y. Stahl, D. Moldt, Distributed P/T net simulation prototypes based on event streaming, in: M. Köhler-Bußmeier, D. Moldt, H. Rölke (Eds.), Proceedings of the International Workshop on Petri Nets and Software Engineering 2024 co-located with the 45th International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2024), June 24 - 25, 2024, Geneva, Switzerland, volume 3730 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2024, pp. 192–216. URL: https://ceur-ws.org/Vol-3730.
[22] L. Clasen, P. Leonhardt, L. Wichelmann, Resilient Distributed P/T Net Simulators, in: [40], 2025. URL: https://ceur-ws.org/Vol-3998.
[23] L. Clasen, C. Nayci, E. Nayci, J. Middendorf, T. Mack, Investigations Towards Dynamic Scaling of Distributed P/T Nets, in: [40], 2025. URL: https://ceur-ws.org/Vol-3998.
[24] L. Clasen, C. Nayci, D. Moldt, Distributed reference net simulation based on event streaming, in: E. G. Amparore, L. Mikulski (Eds.), Application and Theory of Petri Nets and Concurrency - 46th International Conference, PETRI NETS 2025, Paris, France, June 22-27, 2025, Proceedings, volume 15714 of *Lecture Notes in Computer Science*, Springer, 2025, pp. 130–154. URL: https://doi.org/10.1007/978-3-031-94634-9_7. doi:10.1007/978-3-031-94634-9_7.
[25] L. M. Kristensen, L. Petrucci, An approach to distributed state space exploration for coloured petri nets, in: J. Cortadella, W. Reisig (Eds.), Applications and Theory of Petri Nets 2004, 25th International Conference, ICATPN 2004, Bologna, Italy, June 21-25, 2004, Proceedings, Lecture Notes in Computer Science, Springer, 2004, pp. 474–483. URL: https://doi.org/10.1007/978-3-540-27793-4_28. doi:10.1007/978-3-540-27793-4_28.
[26] C. Lakos, L. Petrucci, Modular state space exploration for timed petri nets, Int. J. Softw. Tools Technol. Transf. 9 (2007) 393–411. URL: https://doi.org/10.1007/s10009-007-0033-2. doi:10.1007/S10009-007-0033-2.
[27] M. C. Boukala, L. Petrucci, Distributed model-checking and counterexample search for CTL logic, Int. J. Crit. Comput. Based Syst. 3 (2012) 44–59. URL: https://doi.org/10.1504/IJCCBS.2012.045076. doi:10.1504/IJCCBS.2012.045076.
[28] C. Bellettini, M. Camilli, L. Capra, M. Monga, Distributed CTL model checking using mapreduce: theory and practice, Concurr. Comput. Pract. Exp. 28 (2016) 3025–3041. URL: https://doi.org/10.1002/cpe.3652. doi:10.1002/CPE.3652.
[29] S. Evangelista, L. M. Kristensen, L. Petrucci, Evaluation of a distributed explicit state space exploration algorithm with state reconstruction for RDMA networks, Int. J. Softw. Tools Technol. Transf. 27 (2025) 149–168. URL: https://doi.org/10.1007/s10009-025-00782-5. doi:10.1007/S10009-025-00782-5.
[30] L. Cabac, Modeling Petri Net-Based Multi-Agent Applications, Dissertation, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, 2010. URL: https://ediss.sub.uni-hamburg.de/handle/ediss/3691.
[31] L. Korte, Ein kollaborativer Web-Editor zum Bearbeiten und Simulieren von Renew Referenznetzen, Master thesis, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, 2025.
[32] R. Goud, K. M. van Hee, R. Post, J. M. E. van der Werf, Petriweb: A repository for Petri nets, in: Petri Nets and Other Models of Concurrency-ICATPN 2006: 27th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency, Turku, Finland, June

26-30, 2006. Proceedings 27, Springer, 2006, pp. 411–420.

[33] L. M. Hillah, F. Kordon, Petri nets repository: a tool to benchmark and debug Petri net tools, in: Application and Theory of Petri Nets and Concurrency: 38th International Conference, Petri Nets 2017, Zaragoza, Spain, June 25–30, 2017, Proceedings 38, Springer, 2017, pp. 125–135.

[34] Oracle, The jarsigner command, 2025. URL: https://docs.oracle.com/en/java/javase/21/docs/specs/man/jarsigner.html, accessed: 2026-04-11.

[35] Microsoft, Authenticode digital signatures, 2026. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/install/authenticode, accessed: 2026-04-11.

[36] G. C. Necula, Proof-carrying code, in: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '97, Association for Computing Machinery, New York, NY, USA, 1997, p. 106–119. URL: https://doi.org/10.1145/263699.263712. doi:10.1145/263699.263712.

[37] J. Davies, D. M. German, M. W. Godfrey, A. Hindle, Software bertillonage, Empirical Software Engineering 18 (2013) 1195–1237. URL: https://doi.org/10.1007/s10664-012-9199-7. doi:10.1007/s10664-012-9199-7.

[38] JFrog, What is software provenance?, 2026. URL: https://jfrog.com/learn/grc/software-provenance/, accessed: 2026-04-11.

[39] A. Brodzik, W. Mazurczyk, Ai model signing for integrity verification, in: 2025 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit), 2025, pp. 757–762. doi:10.1109/EuCNC/6GSummit63408.2025.11036922.

[40] M. Köhler-Bußmeier, D. Moldt, H. Rölke (Eds.), Proceedings of the International Workshop on Petri Nets and Software Engineering 2025 co-located with the 46th International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2025), June 22 - 27, 2025, Paris, France, volume 3998 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2025. URL: https://ceur-ws.org/Vol-3998.