accepted for: CARDIS 04, Sixth Smart Card Research and Advanced Application IFIP Conference
22.-27. August 2004, Toulouse, France, as part of the IFIP World Computer Congress

CHECKING AND SIGNING XML DOCUMENTS ON JAVA SMART CARDS

Challenges and Opportunities

Nils Gruschka, Florian Reuter and Norbert Luttenberger
Christian-Albrechts-University of Kiel

Abstract: One major challenge for digitally signing a document is the so called “what you see is what you sign” problem. XML as a meta language for encoding semistructured data offers new opportunities for a solution. The possibility for checking fundamental properties of XML-encoded documents (well-formedness, validity) can be used to improve the security of the signing process for such documents. In this paper we present an architecture for checking and signing XML documents on a smart card in order to enhance the control over the documents to be signed. The proposed architecture has successfully been used to implement a secure, smart card based electronic banking application for the financial transactions system FinTS.

Key words: Java smart cards, XML, digital signature, XML Schema, electronic banking

1. INTRODUCTION

Smart card assistance for generating digital signatures is current state of the art and best practice. This is mainly due to the fact that smart cards nowadays have enough processing power to produce digital signatures for documents by on-card resources (processor and memory) only. This way the owner’s private signing key never has to leave the smart card: The signing key is and remains permanently stored in a tamper-proof environment.

A closer look at the signing process however reveals a still existing major security problem: the problem known as the "what you see is what you sign" problem. Before signing a document the signer usually wants to check the document's syntactic and semantic correctness.

When compared to the traditional process of signing a paper document with a handwritten signature, the difference can easily be identified: In the traditional case, it is relatively easy for the user to assert the correctness, because syntactic and semantic document checking and signature generation are in immediate context. Digitally signing an electronic document is completely different, because checking and signature generation are executed in two different environments, exposing fundamentally different characteristics—different with respect to security on the one hand and processor, memory, and display resources on the other hand.

Traditionally, the signing application computes the document's digest using a one-way hash function and sends the result to the smart card. The card encrypts the digest by an asymmetric cipher using the signing key stored on the card. The resulting value is the digital signature of the document. It is sent back to the signing application. The user can neither check the syntactic correctness of the document (in case of XML documents: well-formedness, validity) nor the semantic correctness of the document. What really is signed is beyond the user's control. It might—for instance—be the digest for a manipulated document. Even if the smart card can be regarded as tamper-proof, the terminal (e.g. a PC) and the programs running on it are vulnerable to viruses and Trojan horses [3, 9, 10]. Such evildoers might obviously also affect signing applications and let them produce valid signatures for—from the user's perspective—invalid documents. Such incidents invalidate the signing process in total.

We propose an enhanced architecture which performs checking and signing of XML documents on Java smart cards, called JXCS architecture. The basic idea of JXCS is to shift the syntactic validation and hash value generation from the vulnerable PC to the trusted smart card. Syntactic validation imposes the following challenges and opportunities: Challenging is the need of processing XML documents on resource constraint Java smart cards. The opportunity of the approach is the possibility to perform syntactic and even semantic checks on the XML document in a tamper-proof environment which improves the security of the signing process.

We propose the need for three major checks on the XML documents to be signed: Well-formedness, validity and content acknowledgement using a class 3 card reader. Taken together all three checks can defeat "what you see is what you sign" attacks.

The paper is organized as follows: Chapter 2 introduces relevant XML fundamentals. The checks are explained more detailed in chapter 3. Chapter 4 then proposes the JXCS architecture. Chapter 5 shows a sample implementation for this architecture. Finally chapter 6 gives the conclusion.

2. XML PRELIMINARIES

2.1 XML well-formedness

An XML document [11] is said to be well-formed if it contains at least one element and fulfills the well-formedness constraints given in the XML specification. Well-formedness is related to the syntactic format of markup, content, comments, document type definition, and so on, and it also ensures the usage of proper Unicode characters and the specification of their encoding. According to the XML specification every XML processing entity has to check and assert the well-formedness property.

The most important constraint is the logical structure of element tags. The tags must be properly braced, meaning that every start element has a corresponding end element, together forming a unique tree structure. For example `<a><b>Hello</b></a>` is well-formed, while `<a>Good</c>` and `<a><b>Bye</a></b>` is not well-formed.

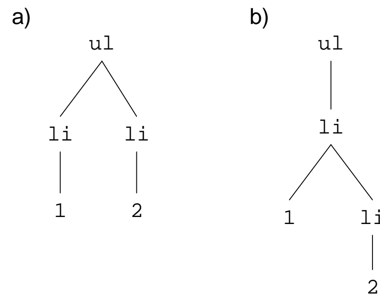

Figure 1: Possible interpretations for the not well-formed XML document: `<ul><li>1<li>2</ul>`

Well-formedness ensures the uniqueness of the XML documents' interpretation. Consider for example the following not well-formed XML fragment: `<ul><li>1<li>2</ul>`. The document allows the following two interpretations shown in figure 1: `<ul><li>1</li><li>2</li></ul>` for a) and `<ul><li>1<li>2</li></li></ul>` for b). In case of well-formedness the interpretation would be unique.

2.2 XML validity

A more restricting property compared to well-formedness of XML documents is its validity. An XML document is said to be *valid* if it is valid with respect to an associated document type declaration or schema. Docu-

ment type definitions resp. schemata express tree grammars, against which the XML documents are checked.

A (regular) tree grammar is a 5-tupel $G = (\Sigma, D, N, P, n_S)$ where,

- $\Sigma = \Sigma_E \cup \Sigma_A$, with Σ_E is a finite set of element types and Σ_A is a finite set of attribute types,
- D is a finite set of data types,
- N is a finite set of non-terminals,
- P is a finite set of production rules of the form $n \rightarrow a(r)$, where $n \in N$ is a non-terminal, $a \in \Sigma$ is a symbol and r is either an element of D or a regular expression over the alphabet N with $L(r) \subset N^*$. Finally
- $n_s \in N$ is the starting non-terminal

Consider for example the following (simplified) FinTS schema fragment

```
<xsd:element name="Amount">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="Value" type="xsd:decimal"/>
      <xsd:element name="Currency" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

defining instances of the form:

```
<Amount>
  <Value>100</Value>
  <Currency>Euro</Currency>
</Amount>
```

The corresponding tree grammar for the above FinTS schema fragment has the following form:

$G = (\Sigma, D, N, P, n_S)$ with $\Sigma = \Sigma_E = \{\text{Amount, Value, Currency}\}$, $D = \{\text{xsd:string, xsd:decimal}\}$, $N = \{N_1, N_2, N_3\}$, $n_S = N_1$ and P containing the following production rules:

- $N_1 \rightarrow \text{Amount}(N_2\ N_3)$
- $N_2 \rightarrow \text{Value(xsd:decimal)}$
- $N_3 \rightarrow \text{Currency(xsd:string)}$

An import property of tree grammars is single-typeness, which means that for each production rule $n \rightarrow a(r)$, non-terminals in its content model

r do not compete with each other. Single-typeness ensures unique parsing. Thus, in the following all tree grammars are assumed to be single-typed [15].

W3C XML Schema offers a rich variety of built-in data types which can be customized using `<restriction>`, `<union>` and `<list>`s. This allows to impose content-related constraints, e.g. the restriction of decimals to an upper bound:

```
<xsd:element name="Value">
  <xsd:restriction base="xsd:decimal">
    <xsd:maxInclusive value="100"/>
  </xsd:restriction>
</xsd:element>
```

2.3 XML Signature

XML Signature is a W3C recommendation [1] for digital signatures using an XML format. It specifies the required XML syntax and the processing rules for creating and representing XML signatures. In the context of the JXCS architecture we used this format and its related processing rules.

```
<Signature>
   <SignedInfo>
       <CanonicalizationMethod/>
       <SignatureMethod/>
       <Reference URI>
          <Transforms></Transforms>
          <DigestMethod/>
          <DigestValue></DigestValue>
       </Reference>
    </SignedInfo>
    <SignatureValue></SignatureValue>
    <KeyInfo></KeyInfo>
  </Signature>
```

Figure 2: XML Signature format (simplified)

An XML Signature is represented by a `Signature` document element which has a structure as given in figure 2. A `Signature` element carries a sequence of three elements: a `SignedInfo` elements, a `SignatureValue` element, and a `KeyInfo` element.

The `SignedInfo` element contains a sequence of a `CanonicalizationMethod` element, a `SignatureMethod` element and one or more `Reference` particles. A `Reference` particle is created for each signed object

(XML document or arbitrary other content, the latter not regarded in this paper). The `Reference` particle includes a reference URI to the data object, which may be found externally or included in the same document. The `Reference` element further contains the digest value for the object (`DigestValue`), and the digest algorithm used (`DigestMethod`). For all signed data objects given in the `Reference` particles, a single signature algorithm element (`SignatureMethod`) and a canonicalization method element (`CanonicalizationMethod`) is available. (Canonicalization e.g. removes redundant whitespaces, sorts attributes, normalizes namespaces, etc.)

The signature value given in the `SignatureValue` element refers to the whole `SignedInfo` element using the signature algorithm and the user's signing key. The user's signing key may be included in the XML Signature in the `KeyInfo` element.

A signature according to the XML Signature recommendation is computed in five steps:

1. canonicalization of the document to be signed,
2. computing the digest of the canonicalized document,
3. generation of the `SignedInfo` element,
4. computation of the signature value,
5. building of the `Signature` document.

2.4 APIs for XML processing

There exist two different types of APIs for XML processing: tree-based APIs like DOM or event-based APIs like SAX [18] or StAX [24]. Tree-based APIs in fact load the complete XML document into main memory which then can be accessed as an XML Infoset [12] like tree. Such kind of APIs put a great strain on system resources, especially if the document is large [18].

Event-based APIs report parsing events directly to the application through callbacks, and do not usually build internal trees. Applications implement handlers to deal with the different events, much like handling events in a graphical user interface.

With respect to the limited resources of a Java s the only useful choice for on card XML processing is an event-based API. We propose a SAX-like event-based API which produces the following events:

- begin-element(*a*),
- end-element(*a*),
- begin-attribute(*a*),
- end-attribute(*a*),
- char-content(*c*),

where $a \in \Sigma$, $c \in$ Unicode*. The above XML document is represented by the following events:

- begin-element(Amount)
- begin-element(Value)
- char-content("100")
- end-element(Value)
- begin-element(Currency)
- char-content("EUR")
- end-element(Currency)
- end-element(Amount)

3. WHY SIGNING XML DOCUMENTS IS DIFFERENT

Why relying on XML for solving the "what you see is what you sign" problem? Our ideas can be summarized in two points:

1. If a document to be signed is either not well-formed in the sense of XML, or not valid in the sense of its accompanying schema, or both, than it must strictly be assumed that the document has been manipulated. In consequence, it has to be dropped, and the user has to notified.
2. A smart card application can extract certain content items for display on the smart card reader¬from a structured and formally described document. The extraction and display operations are fully controlled by the tamper-proof smart card—which is the same environment that generates the digital signature.

The fundamental property of XML documents is well-formedness. According to the XML specification every XML processing entity has to check and assert this property. Regarding digital signing well-formedness is important, since it ensures the uniqueness of the XML documents' interpretation. Well-formedness also ensures the usage of proper Unicode characters and the specification of their encoding. This is also very important regarding digital signatures, since character set manipulation can be used to perform "what you see is what sign" attacks [10].

Validity is a much more restrictive property of XML documents compared to well-formedness. A smart card which checks validity of XML documents with respect to a given schema before signing ensures due to the tamper resistance of the smart card that only certain types of XML documents are signed. Consider for example a smart card which contains your private key, but only signs XML documents which are valid with respect to a purchase order schema. You could give this card to your secretary being

sure, that nothing else than purchase order is signed using your signature. Using additional constrains in the schema, e.g. the restriction of the maximum amount to 100 Euro, eliminates the last chance of misusage.

When operated in a class 3 card reader (i.e. a card reader including a display and a keypad) the card can display selected content and request user confirmation. This finally solves the "what you see is what you sign" problem.

Obviously, XML processing is not an easy task to perform on resource-constraint SmartCards. The following table therefore summarizes the challenging XML properties and the resulting opportunities for improving the signing process:

Table 1: Challenges and opportunities in signing XML documents

Challenge	**Opportunity**
Check well-formedness property on smart card	Confidence that only XML documents are signed
Check validity property on smart card	Confidence that only documents of a certain type are signed
Check validity property and additional content-related constraints (W3C XML Schema simple types) on smart card	Confidence that certain „lower bounds“ on document content are observed (e.g. financial transaction not going over some critical amount)
Display selected content items on SmartCard reader	Confidence that even a manipulated document couldn’t do any harm

4. JXCS SMARTCARD SIGNING ARCHITECTURE

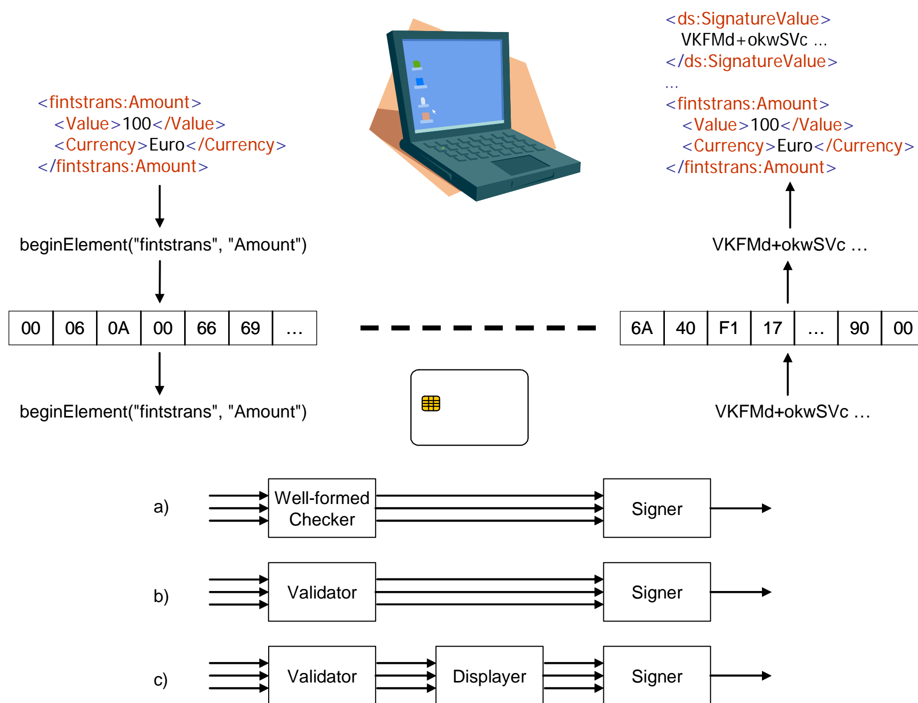

Figure 3: JXCS Architecture Overview

The JXCS architecture offloads all security-critical processing tasks to the tamper-proof smart card environment including:

- well-formedness and validity checking,
- document canonicalization and hashing, and
- signature value computation.

Figure 3 gives an overview over the JXCS architecture. The terminal runs an XML parser which analyzes the document to be signed. The parsing events are coded into APDUs and sent to the smart card. The events are forwarded to a chain of event handlers. These handlers process the total XML document sequentially event by event. Different handlers can be used in this chain according to the specific requirements (see for example figure 3, a) to c)). The chain normally contains event handlers for checking XML document properties like well-formedness or validity, and of course a handler for creating the signature. If an event causes one of the checks to fail, instantly an exception is thrown. If all checks on all events are successful, the signature value of the document is finally returned. From this value the XML Signature is created by the terminal computer. The following paragraphs describe the JXCS architecture handlers in detail.

4.1 Signing Handler

The signing handler's purpose is to digest the canonicalized document, to create the `SignedInfo` element and to compute the `SignatureValue`.

For the purpose of digesting the signing handler reconstructs the canonicalized document from the received parsing event; there is no need to send the original document to the smart card. This approach—based on a single event stream—enables the efficient combination of both checking and signing XML documents on resource constraint smart cards.

Table 2 shows how the canonicalized document is reconstructed from parsing events.

Table 2: Reconstructing the document from events

Event	Parameter 1	Parameter 2	Canon. Document
beginDocument			
beginElement	*name*		<*name*
beginElement	*prefix*	*name*	<*prefix*:*name*
addNamespace	*uri*		xmlns =”*uri*”
addNamespace	*prefix*	*uri*	xmlns:*prefix*=”*uri*”
beginAttribute	*name*		*name*=”
beginAttribute	*prefix*	*name*	*prefix*:*name*=”
endAttribute			“
charContent	*data*		*data*
endElement	*name*		</*name*>
endElement	*prefix*	*name*	</*prefix*:*name*>
endDocument			

For each reconstructed document element the hash generator of the signing handler is triggered which that way digests the whole document piece by piece. Most hashing functions (like MD5 or SHA-1) do not store the whole input data, but process the input stream block by block and only need a small additional digest buffer. Thus the hashing “piece by piece” is very space saving.

If the document has been completely parsed and transmitted to the card, the endDocument event is triggered and the signing object starts the actual signing process. The signer creates the `SignedInfo` (see above) element including the signature, canonicalization and digest algorithm. Furthermore the document's digest and the reference (created from the document's root element) is inserted.

The resulting SignedInfo fragment is signed using the user's private key, stored on the smart card. The resulting value is used to create the entire XML Signature.

As parsing events and the canonicalized document are semantically equivalent, the canonicalized document can be reconstructed from the pars-

ing events, and the digital signature can be generated from the reconstructed document, and the signature is valid only for the parsed document and not for any other document.

4.2 Well-Formedness Checker

The following algorithm is used in the JXCS architecture to check well-formedness:

begin-element(a): Push a on the stack
end-element(a): Let a' be the stack's topmost element.

- If $a = a$', pop a' from stack.
- Otherwise throw a not-wellformed exception.

begin-attribute(a): do nothing
end-attribute(a): do nothing
char-content(c): check if all characters in c are allowed characters

4.3 Validity Checker

The validity of XML documents with respect to a tree grammar (i.e. a document type definition or a schema) can easily be checked by the following algorithm using the proposed event-based API and a stack. The algorithm starts with an empty stack.

begin-element(a):

- If the stack is empty, find a production rule $n_s \rightarrow a(r)$ and push it onto the stack.
- Otherwise let $n' \rightarrow a'(r')$ be the stack's topmost production rule. Search the production rule $n \rightarrow a(r)$ where $n \in \text{next}(r')$.
  - If none such exist, throw a not-valid exception.
  - Otherwise change the actual production $n' \rightarrow a'(r')$ to $n' \rightarrow a'(n' \backslash r')$ and then push $n \rightarrow a(r)$ onto the stack.

end-element(a): Let $n' \rightarrow a'(r')$ be the stack's topmost production rule.

- If $a \neq a'$, throw a not-wellformed exception.
- Otherwise
  - If $\varepsilon \notin L(r')$, throw a not-valid exception.
  - Otherwise pop the production rule from the stack.

begin-attribute(a), end-attribute(a): same as for end-element

char-content(c): Let $n \rightarrow a(r)$ be the stack's topmost production rule.

- If $r \notin D$, throw a not-valid exception.
- If $c \notin r$, throw a not-valid exception.

The set next(r) = $\{a \in \Sigma \mid$ there exists a $\sigma \in \Sigma^*$ with $a\sigma \in L(r)\}$ describes the set of letters accepted next and $x\backslash a$ describes the derivative of x with respect to a, i.e. the expression which remains from a after parsing x. Both next(r) and $x\backslash a$ can be computed efficiently using a simple lookup table [25]. The above algorithm is correct, due to the fact that for a single type tree grammar the next non-terminal can be chosen uniquely [15, 17].

We use a compressed encoding of the production rules based on adjacency lists. Here the running time is $O(m\ n)$ with m is the length of the largest adjacency list. For a fixed tree grammar m is constant, so the running time is still $O(n)$ for a fixed schema or document type definition.

The space consumption is $O(h)$ where h is the height of the parsed tree. For non-recursive tree grammars the maximum height of an XML document is fixed, so the space consumption is limited for a fixed tree grammar. Therefore the memory consumption of the algorithm will not exceed the smart card's resources.

The adjacency list representation for an XML schema is generated off-card and installed on the card. A card can handle one or more schemata.

4.4 Visual inspector for Class 3 Reader

A smart card reader with own display and numerical keyboard (often called "class 3 reader") offers further possibilities for checking the document before signing it. The data exchanged between card reader and smart card can neither be read nor changed by the terminal or any Trojan horse running on it. So unless the reader's firmware has not been modified, the card reader can be used as a secure display for the information send to the card.

A simple solution to the "what you see is what you sign" problem would be showing the complete document on the display prior to sending it to the smart card. This would indeed be useless for at most all practical purposes. These displays normally only have 1 to 3 rows with at most 20 characters each. Most users' acceptance for viewing a complete document on such a display would be very low.

Displaying selected parts of arbitrary XML documents otherwise is ineffective for checking the document to be signed. A single element may have totally different semantics in different contexts. If the user acknowledges e.g. the prompt `<Amount>1000 EUR</Amount>` he does not know the denotation of this element. He could buy a cheap car or transfer the money to the

Nigeria connection. Even if all ancestors of the element are displayed additionally, the semantic is generally not unique.

This is solved by validating the XML document to a specific schema. In this case the context and semantic of an element is unambiguous. If the validator validates the documents e.g. against a schema for cash remittance, the user can be sure to sign a transfer of 1000 Euro, if he acknowledges the element `<Amount>1000 EUR</Amount>`.

The functionality of the display component is the following. When the schema is transferred to the smart card, the most “critical” elements are marked for displaying. The display component reads the content of these elements from the event stream. These values have to be acknowledged by the user. If one is rejected an exception is thrown.

5. FINTS SAMPLE IMPLEMENTATION

5.1 Sample Application: FinTS

FinTS is the newest version of HBCI, the home banking computer interface developed by the German Central Banking Committee (ZKA). It defines a standard for home banking and specifies the relationship between customer products and bank systems. FinTS allows more flexible and convenient online banking than other systems. To ensure secure transactions over open networks, cryptographic functions and smart cards are used.

The actual version’s communication is based completely on XML. All messages exchanged between the FinTS server and the FinTS clients are XML documents. Orders, like cash remittances, are signed using a personal smart card and coded as XML signature. The whole transaction message – containing one or more orders – is encrypted into an XML encryption document.

Banking transactions are obviously extreme security relevant and a profitable target for attacks. The user wants assurance, that the document he creates using his online banking program is actually the one signed by the card and send to his bank.

5.2 Implementation

The proposed architecture was utilized to implement a client with a secure signing process for FinTS banking transactions. We implemented a client program on the terminal creating FinTS remittance transactions and sending it to the smart card. The smart card contains the user’s private key

and a signing component. It also contains a validator, validating the input document against a modified FinTS schema for a remittance. The schema has been altered in the way that the remittance value must be less or equal 100 and the currency must be EUR. Thus the smart card will sign only documents being a valid FinTS remittance with a maximum of 100 Euro. Thus the user has the assurance, that his signing card will never sign anything else, no matter what a Trojan horse would perform to the documents. Additionally the display component let the user acknowledge the remittance's most important content values like target account number. Thus even modifications on these values would be detected.

The sample client program is a GUI that creates a FinTS cash remittance document from the input values. This XML document is parsed into XML events, which are coded as TLV into request APDUs. Due to efficiency reasons not single events are sent to the card. Instead the amount of events fitting into an APDU is transferred, improving performance by up to 20%.

On the Java smart card a main applet and the following event handlers are implemented: validator, displayer and signer. The XML events are decoded from the APDUs by the applet, which calls the event handler's appropriate event methods. Once an exception is thrown by of the handlers, an error code is returned to the client. In order to simplify the implementation the algorithms for hashing, signing, canonicalization are set statically. The final application needs approximately 13000 bytes plus the binary schema representation. The signing of a typical cash remittance needs approximately 15 seconds.

The validator is an implementation of the above described validity checking algorithm. In our sample application we provided the validator on the smart card with the FinTS schema. The original schema has 5537 bytes, the binary tree grammar only just 954 bytes. As shown above the stack's size is limited for a fixed schema. The stack can be implemented as list of fixed length. Thus dynamic object instantiating can be avoided, which is critical for a Java card without garbage collection. For the FinTS schema the maximum size needed for the validator's stack is 200 bytes.

The display component will be configured when uploading the schema. The simple type elements, that shall be displayed, are specified by an XPath like expression. The displayer collects the content for these elements from the charContent events. He then waits for the acknowledgement for these values from the card reader. Due to the stringent master-slave-relationship between the terminal and the smart card, the card can not request the card reader for displaying these values. Instead, the PC sends the to-be-acknowledged values to the card reader contained in a special command. This command instructs the reader to display the value and send it to the card if it is acknowledged by the user. The displayer compares these values

with the content extracted from the events. If this fails an exception is thrown.

As pointed out above the signing component creates the `Signature` fragment from the events. In order to minimize the communication between PC and card, not the total `Signature` fragment is returned to the client but only the documents digest and the signature of the `SignedInfo` fragment. As the algorithms are fixed, the client program can create the same `Signature` fragment from the digest and the signature values. And of course this is no lack of security.

The Java smart cards used are JCOP 21id from IBM. These cards are compliant to JavaCard 2.1.1, OpenPlatform 2.0.1' and FIPS 140-2 level 3. They have 30 Kbytes EEPROM as persistent Java heap, 590 bytes RAM as transient Java heap and 200 bytes RAM as Java stack. The card reader used is a class 3 reader (2 x 16 character display, numerical keypad) from ReinerSCT.

6. CONCLUSION AND FUTURE WORK

In this article we have shown how processing XML documents on a smart card arises new opportunities for signing them. By checking properties like well-formedness and validity the users gains more control over the documents signed using his private key. We have also shown, that checking a document's validity according to a specific grammar, allows showing single elements from the document on a reader's display without losing the element's semantic. This way we can greatly improve the security of the signing process for XML document and even approach a solution for the "what you see is what you sign" problem.

Future research will focus on applying the XML processing smart card technology on XML communication protocols to improve e.g. Web Service security.

7. ACKNOWLEDGEMENT

We would like to thank Karsten Strunk and Jesper Zedlitz for implementing the secure FinTS signing client including an interface to a FinTS server system. We also like to thank Christian Friberg and Rainer Segebrecht of PPI Financial Services Kiel for supporting the implementation process.

8. REFERENCES

[1] Mark Bartel et al. *XML-Signatur Syntax and Processing – W3C Recommendation 12 February 2002*. W3C (World Wide Web Consortium), 2002.

[2] John Boyer. *Canonical XML, Version 1.0 – W3C Recommendation 15 March 2001*. W3C (World Wide Web Consortium), 2001.

[3] Armin B. Cremers, Adrian Spalka, and Hanno Langweg. The Fairy Tale of 'What You See Is What You Sign' – Trojan Horse Attacks on Software for Digital Signatures. In *IFIP Working Conference on Security and Control of IT in Society-II (SCITS-II*), Bratislava, Slovakia, June 2001.

[4] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644-654, November 1976.

[5] Henry S. Thompson et al. *XML Schema Part 1: Structures – W3C Recommendation 2 May 2001*. W3C (World Wide Web Consortium), 2001.

[6] Paul V. Biron and Ashok Malhotra. *XML Schema Part 2: Datatypes – W3C Recommendation 2 May 2001*. W3C (World Wide Web Consortium), 2001.

[7] James Clark, Steve DeRose. *XML Path Language (XPath) – W3C Recommendation 16 November 1999*. W3C (World Wide Web Consortium), 2001.

[8] Tim Redhead and Dean Povey. The Problem with Secure On-Line Banking. In *Proceedings of the XVIIth annual South East Asia Regional Conference (SEARCC'98)*, July 1998

[9] Arnd Weber. See What You Sign. Secure Implementation of Digital Signatures. In *Intelligence in Services and Networks: Technology for Ubiquitous Telecom Services* (IS&N'98), Springer-Verlag LNCS 1430, 509-520, Berlin, 1998.

[10] Audun Jøsang, Dean Povey, and Authony Ho. *What You See is Not Always What You Sign*. AUUG 2002 - Measure, Monitor, Control, September 2002

[11] Tim Bray et al. *Extensible Markup Language (XML) 1.0 (Third Edition) W3C Recommendation 04 February 2004*. W3C (World Wide Web Consortium), 2004.

[12] John Cowan, Richard Tobin. *XML Information Set (Second Edition) W3C Recommendation 4 February 2004*. W3C (World Wide Web Consortium), 2004.

[13] P. Buneman. *Semistructured data*. Tutorial in Proceedings of the 16th ACM Symposium on Principles of Database Systems, 1997

[14] Hiroshi Maruyama et al. *XML and Java: developing Web applications*. Pearson Education. 2nd ed. 2002.

[15] Makoto Murata, Dongwon Lee, and Murali Mani. *Taxonomy of XML Schema Languages using Formal Language Theory*. Extreme Markup Languages 2000, August 13-14, 2000. Montreal, Canada.

[16] Boris Chidlovskii. *Using Regular Tree Automata as XML Schemas*. IEEE Advances in Digital Libraries 2000 (ADL 2000). May 22 - 24, 2000. Washington, D.C.

[17] F. Neven. *Automata theory for XML researchers*. SIGMOD Record, 31(3), 2002.

[18] The SAX Project, URL: http://www.saxproject.org/

[19] IBM JCOP embedded security software. URL: http://www.zurich.ibm.com/jcop/

[20] Sun Microsystems: JavaCard 2.1.1 http://java.sun.com/products/javacard

[21] Global Platform Consortium: OpenPlatform 2.0.1'. URL: http://www.globalplatform.org/

[22] FIPS PUB 140-2: Security Requirements For Cryptographic Modules, May 2001. URL: http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf

[23] FinTS Financial Transaction Services 3.0. URL: http://www.fints.org/

[24] JSR 173: Streaming API for XML. Java Community Process.

[25] Janusz A: Brzozowski. *Derivatives of regular expressions*. Journal of the ACM, 11(4), 1964.