# Analysis of Signature Wrapping Attacks and Countermeasures

Sebastian Gajek, Meiko Jensen, Lijun Liao, and Jörg Schwenk
*Horst Görtz Institute for IT Security, Ruhr University Bochum, Germany*
{`Sebastian.Gajek|Meiko.Jensen|Lijun.Liao|Joerg.Schwenk`}`@ruhr-uni-bochum.de`

## Abstract

*In recent research it turned out that Boolean verification of digital signatures in the context of WS-Security is likely to fail: If parts of a SOAP message are signed and the signature verification applied to the whole document returns true, then nevertheless the document may have been significantly altered.*

*In this paper, we provide a detailed analysis on the possible scenarios that enable these signature wrapping attacks. Derived from this analysis, we propose a new solution that uses a subset of XPath instead of ID attributes to point to the signed subtree, and show that this solution is both efficient and secure.*

## 1. Introduction

The WS-* family of security protocols [1] sketches a security framework that addresses many of the security issues concerning Web Services. However, this strong security framework is built on weak foundations: McIntosh and Austel [14] have shown that the content of a SOAP message protected by an XML Signature [3] as specified in WS-Security [16] can be altered without invalidating the signature. This so-called *wrapping attack* or *XML rewriting attack [7], [20]* is possible because the referencing schemes used to locate parts of a SOAP message document differ between the signature verification function and the application logic.

We investigate the possible scenarios of the two most commonly used XML referencing schemes, ID referencing and XPath expressions, and show (by constructing realistic counterexamples) that wrapping attacks are still feasible for many types of these expressions.

We propose to use a subset of XPath, called FastXPath, instead of ID attributes for signature referencing in Web Services messages. If the proposed FastXPath is used, this protects against wrapping attacks in most reasonable scenarios without causing the performance impact associated with the use of complex XPath expressions.

The rest of this paper is structured as follows: Initially, we provide some technical background (Section 2). Then, we explain signature wrapping attacks in Section 3, and discuss related work in Section 4. In Section 5, we provide an in-depth analysis of the different scenarios that may occurr in real-world applications, and we analyze their resistance to signature wrapping attacks. Then, we describe the proposed FastXPath referencing scheme (Section 6), and finally, we conclude the paper in Section 7.

## 2. Technical Background

### 2.1. XPath Filtering

The signature transform XPath Filtering [3] is developed to select complex node sets based on the XPath specification [6]. The XPath Filtering expression is a Boolean expression, unlike the standard XPath expression.

This approach of using XPath raises the likelihood of misinterpretation. For the example of Fig. 1 the expression `/Envelope/Header/Shipping/Departure` results including the whole document, and `Envelope/Header/Shipping/Departure` results that no node of the document is included.

This type of misinterpreting the XML signature standard is rather common, see e.g. the WS-I basic security profile [15, Section 8.2], and papers about wrapping attack [14]. Due to the complexity of the expressions in XPath Filtering, 1) it is difficult to define a correct expression, 2) all nodes must be traversed and evaluated according to this expression (which drastically impacts on performance), and 3) it increases the security problem (generally raising complexity is accompanied by weakening security).

### 2.2. XPath Filter 2

The XPath Filter 2 transform is introduced [8] in order to cope with these usability issues, and also to improve transformation performance. It is designed to specify a subset of a given XML document as the information content to be signed. Unlike XPath Filtering, which was based on evaluating boolean-result expressions on all document nodes, XPath Filter 2 uses the standard XPath expression as defined in the XPath standard. XPath Filter 2 allows a sequence of XPath expressions to select node sets, these sets are then combined using set intersection, subtraction, and union, according to the attribute `Filter`. A detailed example can be found in [8, Section 4].

Since XPath Filter 2 allows arbitrary XPath expressions and three filter types, it may become hard to define a correct XPath Filter expression. If multiple XPath expressions with different filter types are applied, it is hard to determine whether a node and all its descendants are included or excluded. Hence most implementations traverse all nodes in the tree, resulting in a still suboptimal evaluation performance.

## 3. Signature Wrapping Attacks

Wrapping attacks aim at injecting a faked element into the message structure so that a valid signature covers the unmodified element while the faked one is processed by the application logic. As a result, an attacker can perform an arbitrary Web Service request while authenticating as a legitimate user.

Fig. 1 illustrates the structure of a SOAP message as created by the sender. WS-Security [16] is used to authenticate the departure time and the manifest for the container with the identifier `"mf1"`. An XML Signature protects the whole `<Manifest>` element and `<Departure>` element by canonicalizing, hashing and digitally signing them (illustrated with gray color). The `<Manifest>` element to be protected in this way is referenced by an `"Id"` attribute with the value `"mf1"`, and the `<Departure>` element is referenced by the XPath expression `"//Shipping[1]/Departure"`.

Upon mounting the wrapping attack, the `<Manifest Id ="mf1">` element is moved to a position within the SOAP header unknown to the application logic, resulting in that the logic never processes this element. In our example, this is done by embedding a new `<Wrapper>` element into the SOAP header and adding `<Manifest Id ="mf1">` as a child (preserving all its descendants). Additionally, a new `<Manifest Id ="newMf1">` element is added at the original position, having correct structure but different content. The modified message is depicted in Fig. 2. Those parts of the tree that are processed by the signature verification function are colored in gray, those parts processed by the application logic for `CId "c1"` are contained within `/Envelope/Body/Manifest[@CId="c1"]` and `/Envelope/Header/Shipping[@CId="c1"]/Departure`. For convenience, we omitted prefixes and namespaces in the text.

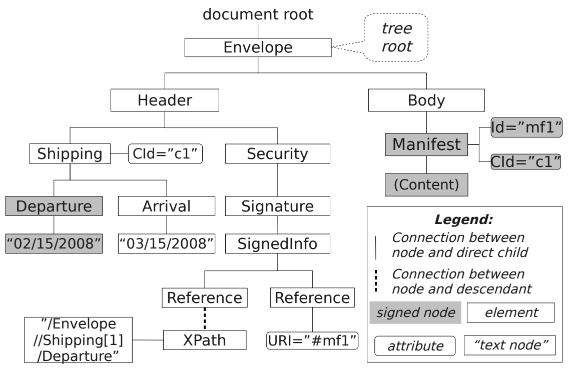

Figure 1. SOAP message with XML signature

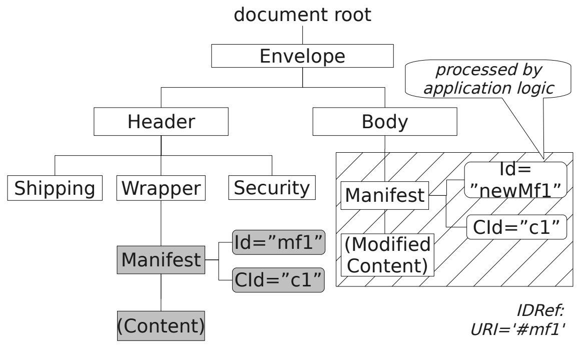

Figure 2. modified SOAP message with valid signature (IDRef)

The signature verification function finds the element with the attribute `"Id"` of value `"mf1"`. Since the content of the element is not modified, the signature remains valid. However, the application logic processes the `<Manifest Id ="newMf1">` element, what results in that the attacker succeeds to invoke an arbitrary service operation with arbitrary parameters, executed in the context of the user's authentication.

The reason for the exploit is that signature verification and application logic process different elements, because they use different referencing methods to locate them.

## 4. Related Work

### 4.1. Policy Approach

McIntosh and Austel [14] show how to protect against certain wrapping attacks by improving the security policy to be followed by sender and receiver.

On the other hand, they also show how to counterfeit each new security policy by a new, more sophisticated wrapping attack.

In addition, the complex security policies employed are not presented in XML syntax, thus they have to be hardcoded into the application. By doing so, one would loose all advantages of service-oriented architectures, because services can no longer be loosely coupled.

### 4.2. Inline Approach

Rahaman, Schaad and Rits [20], [18], [19] propose the inline approach for early detection of wrapping attacks. Our previous work [11] demonstrates that this approach is still vulnerable to wrappping attacks.

Benameur, Kadir, and Fenet [5] extend the inline approach as follows: 1) adding an element that contains the depth information of the signed object; 2) keeping other information besides the name of the signed object's parent; or 3) identifying the parent of the signed object by adding an `"Id"` attribute and keeping its value and the name of its parent.

This extension is still vulnerable, because one can move the signed element together with its parent to somewhere so that the depth of the signed element is not changed.

The weakness of the (extended) inline approach is that the SOAP Account only preserves the relationship to its parent and sibling elements. This is a relative position in the DOM tree, which is not bound to a fixed location information (such as the document root node). Thus, it is still possible to move the whole structure that is protected by the SOAP Account information to a new position, what lets wrapping attacks remain possible.

To mitigate the vulnerability, our recommendation is to consider the absolute path from a signed element to the document's root element ("vertical fixing") and to its siblings ("horizontal fixing"). The position of the signed element must be fixed so that it is infeasible to move the signed element without invalidating the reference—and thus the signature. This is discussed in detail in Section 5.

### 4.3. Modified Verification Functions for XML Signature

In previous work [11], we proposed a complementary solution that relies on changing the signature verification function to return more than just a Boolean value. This way, wrapping attacks may be detected by the application logic in one of two ways: 1) the signature verification function returns a structure-based position indicator. This indicator helps the application logic to determine whether the processed nodes are signed. However, this obviously may lead to interoperability issues; 2) the signature verification function returns a filter that disallows the application logic to access those contents that are not signed. Thus, the filter prevents the application logic from accessing unsigned contents. This way, wrapping attacks are disabled completely, but in most Web Service messages, direct access to unsigned contents is also required. Thus, this strict document access policy cannot be enforced properly.

In both ways, the signature verification function has to forward the position indicator or signed content filter to the application logic. Though this is likely to disable wrapping attacks in general, it is doubtful that these approaches can be applied to real-world scenarios.

## 5. Signature Wrapping Scenarios

Wrapping attacks show that location information is an essential part of the semantics of XML Signatures. This contrasts to classical cryptographic data formats, such as OpenPGP [9] or PKCS#7 [13].

In case of XML, wrapping attacks exploit the semantics of XML Signatures. The use of an `"Id"` attribute to identify signed content implies a meaning like "if the hash value of the referenced data is the same as within the `<DigestValue>` element, then the signature is valid regardless where the data is located within the base document". If the application logic *expects* the signed data at a certain location, an XML Signature format should be used whose semantics says that "the signature is only valid if it is located at or next to a certain location".

XPath restricts, in a way, the position of the signed content. However, this restriction largely depends on the particular XPath expression. In the following, we analyze the different scenarios, with the signed content referenced by ID attributes or different XPath expressions.

Before we go into the details, we introduce two definitions: the *hashed subtree* and the *protected subtree*. Hashed subtrees are the subtrees that contain only nodes that are input to the hash algorithm. Protected subtrees are subtrees that contain hashed subtrees and other nodes which are needed to locate the hashed subtrees (e.g. siblings that influence the horizontal position). For both definitions, assume that the selected nodeset is the direct input of the hash algorithm, i.e. there is no other XML transform, e.g. XSLT transform, being applied.

## 5.1. Identifier Referencing

Due to its simplicity, Identifier-based referencing is recommended or implied in the standards [2], [15], and it is most widely used. For each ID reference, there always exist equivalent XPath Filtering and XPath Filter 2 expressions. For example, an element with attribute `Id="myId"` can be located either using ID referencing via `URI="#myId"`, with XPath Filtering via `ancestor-or-self::node()[@Id="myId"]`, or with XPath Filter 2 via `//*[@Id="myId"]`.

Since the location of a signed element does not matter, it can be moved anywhere within the searching scope of the signature verification process. The easiest way to exploit this behaviour for performing a wrapping attack is to move the signed element (with all its descendants) to a position within the document so that it can be verified by the signature verification logic but is not processed by the application logic.

In this case, there exists only one hashed and one protected subtree, which are identical. Considering the initial example in Fig. 1 and the ID reference `URI="#mf1"`, an example attack is illustrated in Fig. 2 and discussed in Section 3.

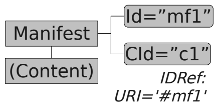

Figure 3. protected subtree (ID referencing)

Considering the original message (Fig. 1) and the modified message (Fig. 2), it can be seen that they have the same hashed subtree and the same protected subtree, as depicted in Fig. 3. That means, the signature's semantics is not violated. However, the actual position of the hashed subtree may not be given any meaning by the application logic.

## 5.2. XPath Referencing with descendant-or-self

In XPath, `//` is the abbreviated syntax for `/descendant-or-self::node()/`, it selects the context node and all its descendants. `//` may lead or occur within any XPath expression. An XPath expression with `//` in the middle, e.g. *path1//path2*, first selects all nodes (denoted as *nodes1*) that satisfy the part before `//`, e.g. *path1*. Based on *nodes1*, it then selects all nodes (denoted as *nodes2*) that satisfy the part after `//`, e.g. *path2*. The positional relation between *nodes2* and *nodes1* does not matter here. For an XPath expression with leading `//`, *part1* is `null`. Thus, *nodes1* is the document root. Without loss of generality, in the following we only consider XPath expressions with `//` in the middle.

At first, we investigate an XPath expression with position predicate in the part after `//`. We assume that the XPath expression `/Envelope//Shipping[1]/Departure` is used to locate the signed content in the message in Fig. 1. Note that in XPath Filter 2 this expression is combined with the `intersect` filter, and in XPath Filtering it is expressed as `ancestor-or-self::Departure[parent::node()=/Envelope//Shipping[1]]`.

This XPath expression selects the `<Departure>` element whose parent is the first `<Shipping>` element within the root element `<Envelope>`. A wrapping attack succeeds if an attacker can 1) replace the original `<Shipping>` element with a new one, and 2) move the original `<Shipping>` element to another location that precedes the modified one within the root element `<Envelope>` (in document order).

The new position should be unknown to the application logic. This modification cannot be detected, because the verification logic here only validates the hash value of the original element. An example attack is depicted in Fig. 4.

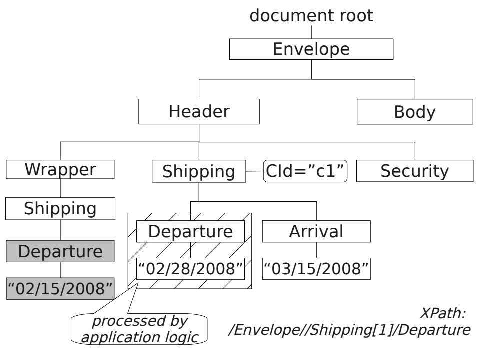

Figure 4. modified SOAP message with valid signature (XPath with // and position predicate)

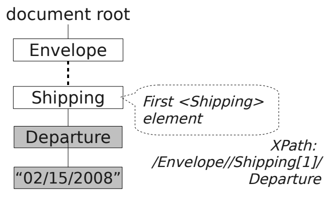

Figure 5. protected subtree (XPath with // and position predicate))

Considering the original message (in Fig. 1) and the modified message (in Fig. 4), they have same hashed

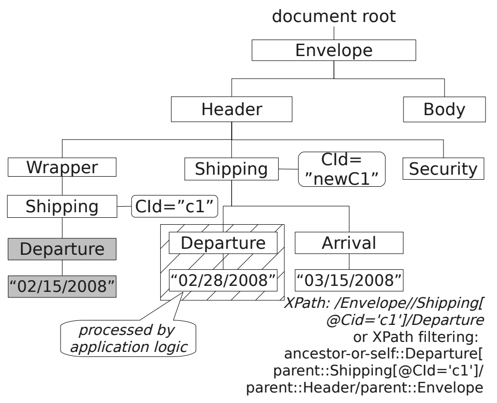
document root
Envelope
Header
Body
Wrapper
Shipping
CId= "newC1"
Security
Shipping
CId="c1"
Departure
Arrival
Departure
"02/28/2008"
"03/15/2008"
"02/15/2008"
processed by application logic
XPath: /Envelope//Shipping[ @Cid='c1']/Departure
or XPath filtering: ancestor-or-self::Departure[ parent::Shipping[@CId='c1']/ parent::Header/parent::Envelope

Figure 6. modified SOAP message with valid signature (XPath with // and attribute predicate)

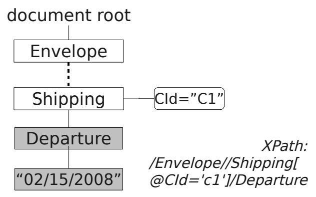

Figure 7. protected subtree (XPath with // and attribute predicate)

subtree and same protected subtree, as depicted in Fig. 5. That means, for the message in Fig. 4, the signature semantics is not violated.

Next, we discuss an XPath expression with attribute predicate in the part after `//`. We assume that the XPath expression `/Envelope//Shipping[@CId="c1"]/Departure` is used to locate the signed content in the message in Fig. 1. Note that in XPath Filter 2 this expression is combined with the `intersect` filter, and in XPath Filtering it is expressed as `ancestor-or-self::Departure[parent::node()=/Envelope//Shipping[@CId="c1"]`.

This XPath expression selects the `<Departure>` element whose parent is the `<Shipping>` element that contains the attribute `CId="c1"`, within the root element `<Envelope>`. A wrapping attack is successful if an attacker can 1) replace the original `<Shipping>` element with a new one without the attribute `CId="c1"`, and 2) move the original `<Shipping>` element somewhere unknown to the application logic. An example attack is depicted in Fig. 6.

Considering the original message (in Fig. 1) and the modified message (in Fig. 6), they have same hashed subtree and same protected subtree (in Fig. 7). That means, the signature semantics is not violated.

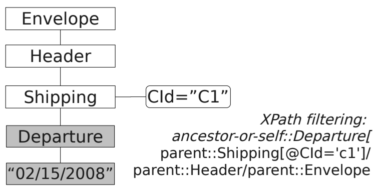
Envelope
Header
Shipping
CId="C1"
Departure
"02/15/2008"
XPath filtering: ancestor-or-self::Departure[ parent::Shipping[@CId='c1']/ parent::Header/parent::Envelope

Figure 8. protected subtree (relative XPath)

### 5.3. Relative XPath Referencing with XPath Filtering

Some applications may want to sign elements with special ancestors. In such situations, XPath Filtering can be applied. For example, if one wishes to sign the `<Departure>` element whose parent is `<Shipping CId="c1">` in the message in Fig. 1, the signed content can be filtered by the XPath Filtering expression `ancestor-or-self::Departure/parent::Shipping[@CId="c1"]/parent::Header/parent::Envelope`.

Note that it is impossible to specify such relative references in XPath Filter 2, since, in XML signature, XPath Filter 2 is applied to filter the whole referenced document. Thus, the context node is the document root. In this sense, a leading `/` is implicitly added to the relative XPath expression.

A wrapping attack is successful if an attacker can 1) replace the original `<Shipping>` element with a new one without the attribute `CId="c1"`, and 2) move the original `<Shipping>` element somewhere unknown to the application logic. The attack in Fig. 6 can also be applied here.

Considering the original message (Fig. 1) and the modified message (Fig. 6), they have the same hashed subtree and the same protected subtree, as depicted in Fig. 8. That means, the signature semantics is not violated.

### 5.4. Absolute XPath Referencing without descendant-or-self

An absolute XPath consists of `/`, optionally followed by a relative location path. A `/` by itself selects the root node of the document that contains the context node. Note that in XPath Filter 2 the document root is considered as the context node, a relative XPath there is actually considered as an absolute XPath.

In the following, we consider following XPath expressions:

1) `/Envelope/Header/Shipping[1]/Departure` or equivalent in XPath Filtering:

`ancestor-or-self::Departure[parent::node()=/Envelope/Header/Shipping[1]]`

2) `/Envelope/Header/Shipping[@CId="c1"]/Departure` or equivalent in XPath Filtering:
`ancestor-or-self::Departure[parent::node()=/Envelope/Header/Shipping[@CId= "c1"]]`

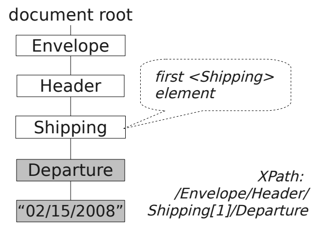

Figure 9. protected subtree (absolute XPath without // but with position predicate)

In the first XPath expression each step of the vertical position of the signed element within the complete document is fixed. Due to the restrictions of SOAP messages, the leading `/Envelope/Header` points to the unique `<Header>` element. Thus, together with the position restriction in step `Shipping[1]`, the horizontal position of `<Shipping>` is also fixed.

By applying XPath expression 1 to the message in Fig. 1, the protected subtree is as depicted in Fig. 9.

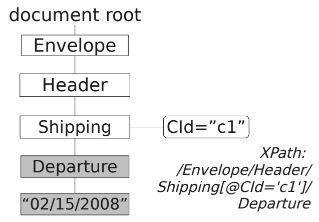

Figure 10. protected subtree (absolute XPath without // but with attribute predicate)

For XPath expression 2, the result is the same, except that the horizontal position of the `<Shipping>` element is fixed by the attribute `CId="c1"`. Fig. 10 shows the protected subtree that results from applying this XPath expression to the message in Fig. 1.

Assume that the application logic processes according to one of the following rules: 1) the `<Departure>` whose parent is the first `<Shipping>` within `/Envelope/Header`, or 2) the `<Departure>` whose parent is the `<Shipping>` (with the attribute `CId="c1"`) within `/Envelope/Header`.

Then, the XML signature with the protected subtree in Fig. 9 is *not* vulnerable to wrapping attacks if *rule 1* is applied. Similarly, the XML signature with the protected subtree in Fig. 10 is *not* vulnerable if *rule 2* is applied.

## 6. FastXPath: Structure-based Referencing

Resulting from the considerations discussed above, our approach to fend wrapping attacks is to fix the vertical (and maybe also the horizontal) position of the signed elements by using an absolute XPath without wildcards (i.e. starting at the document root). This way, it becomes nearly impossible to successfully perform a signature wrapping attack, as any relocation of the signed contents immediately results in an invalidation of the signature value. Nevertheless, this benefit comes with some costs regarding flexibility, as there might be some real-world scenarios where relocation of signed contents may become necessary. However, note that whenever it is possible to move a signed content within an XML document without invalidating the signature, this can always be performed by both the users and the attackers.

In the past, a major argument against the XPath approach is the weak performance of the XPath transform. Indeed, for evaluating XPath expressions it usually becomes necessary to parse the XML document into a DOM tree representation, which the XPath expression can be evaluated against. This overhead is increased by the way XPath was used in the early XPath transforms stated in the XML signature standard, which involved evaluating a certain XPath expression at every node of the DOM tree. This use of XPath caused a severe performance killer for applying the XPath transform in Web Services scenarios.

```
FastXPath      ::= '/' RelativeFastXPath
RelativeFastXPath ::= Step
             | RelativeFastXPath '/'Step
Step        ::= QName PredicatePosition?
PredicatePosition::= Position Predicate?
                 | Predicate Position?
Position       ::= '[' [1-9][0-9]* ']'
Predicate      ::= '[' PredicateExpr ']'
PredicateExpr  ::= PredicateStep
     | PredicateExpr 'and' PredicateStep
PredicateStep  ::= '@' QName '=' Literal
Literal        ::=  '"' [^"]* '"'
                  | "'" [^']* "'"
```

Figure 11. BNF definition of FastXPath.

In order to cope with this issue, XPath Filter 2 [8] has been introduced, but solely to improve usability and performance, not security. Other approaches tried

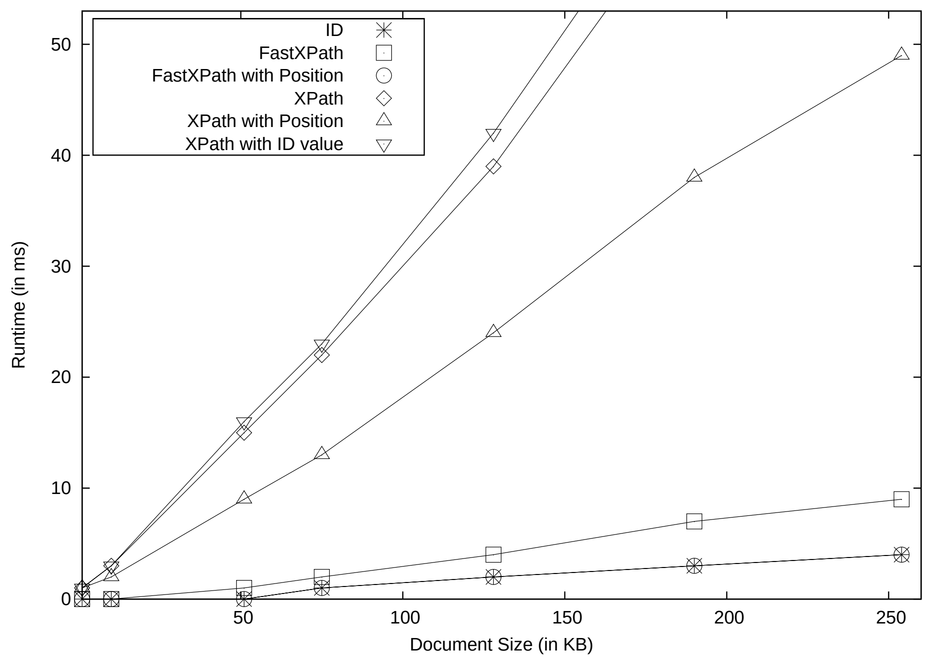

Figure 12. Runtime comparison of ID, ForwardXPath, and full XPath referencing

to re-engineer arbitrary XPath expressions to match a certain subset of XPath, which can then be evaluated faster [12], [4], [17], but all of these pose some restrictions to the usage scenario, and none copes with the signature wrapping threat.

Thus, to simplify and secure the use of XPath in XML digital signature, we define an even simpler subset of XPath, called FastXPath, in Fig. 11. In that definition, `^"` and `^'` stand for all letters except `"` and `'`, respectively.

FastXPath contains only forward directions, both in the Axis and in the Predicate. The Predicate contains Boolean expressions connected by "and". Wildcards may be used, but it can be shown that their use would enable wrapping attacks. Additionally, every FastXPath expression must start with "/" which indicates that the evaluation always starts at the document root.

Some legal FastXPath expressions are e.g. `/Envelope/Body`, and `/Envelope/Header/Security[1][@role="next"]`. Some illegal FastXPath expressions are `//Body`, `/Envelope/Header/Security[role="next" or mustUnderstand="true"]`.

FastXPath allows for fast single-pass SAX style selection of the elements to process. The program checks whether the path starts with '/', and then checks the current node and the current path step (in the next program step, the current node is replaced by its child, and the current path step is replaced by the next one). This way, the evaluation overhead can be reduced drastically compared to full XPath evaluation.

For our evaluation, we have implemented ID-based and FastXPath referencing using the stream-based XML parser StAX. The full XPath referencing—like XPath Filtering or XPath Filter 2—in the evaluation uses the DOM-based XML parser Xerces. Note that it is very difficult to use a non-DOM-based XML parser for full XPath. In the evaluation, the referenced element was always located at the exact middle of the document. The evaluation was performed using SUN JRE 1.6.0 on a PC with 2.8 GHz Pentium 4 CPU and 2.5 GB memory.

As our evaluation shows (see Fig. 12), the performance of FastXPath is equal or at least comparable to ID-based referencing approaches, and outperforms any full XPath referencing by far. As can be seen in the figure, the performance of FastXPath and ID referencing is equivalent if the FastXPath expression also provides position indicators for all its steps (e.g. `/Envelope[1]/Header[1]/Shipping[2]/Departure[1]`), and stays within a slow-down factor of 2.5 if these are missing. Thus, the common argument against tree-based referencing approaches applies to full XPath only, and can be circumvented by using a sensible limitation like FastXPath.

On the other hand, FastXPath turns out to be best possibly resistant to signature wrapping attacks. As it requires to explicitly name every single element on the path from document root to the signed subtree, there is no flexibility here to move any signed contents to another location within the document. Additionally, as the FastXPath expression can also include any ID attribute, it securely supports both the ID-based and the structure-based XML access method, as both methods will point to the same element. Thus, there is no possibility for deviation between the signature verification function's access method and the application logic's access method.

However, even with FastXPath the threat of signature wrapping attacks cannot be averted completely. If the protected content is as depicted in Fig. 9 and rule 2 is applied in the application logic, a wrapping attack is still possible. It is similar for Fig. 10 and rule 1. Hence, there may be no default solution that solves all problems related to XML signature references, but to our consideration these scenarios are rare in real-world applications.

Though FastXPath poses severe restrictions to the abilities of defining a signature reference, this turns out to be a trade-off between security and flexibility, as every flexibility within the reference can potentially be exploited for a wrapping attack. To our consideration, the proposed limitations on the reference abilities are not posing a big problem to real-world scenarios, but the opposite option probably would.

## 7. Conclusion

In this paper we have revisited the problem of wrapping attacks. We analyzed the potential scenarios that lead to the signature wrapping vulnerability, and we derived a new solution to the problem. We proposed FastXPath to point to the signed subtree, and showed that it is fulfilling this task in a secure and performant way, disabling signature wrapping attacks for all reasonable scenarios.

Our future work consists in defining a formal semantics for XML Signature elements along with an investigation on the different usage intentions of digital signatures in Web Service messages (e.g. data integrity, authentication etc.). The results are to be processed within a signature application advisor tool.

## References

[1] Security in a Web Services World: A Proposed Architecture and Roadmap, April 7, 2002. http://www.ibm.com/developerworks/library/specification/ws-secmap/.

[2] A. Nadalin, C. Kaler, R. Monzillo, and P. Hallam-Baker (Editor). Web services security: SOAP message security 1.1, Nov. 2006.

[3] M. Bartel, J. Boyer, B. Fox, B. LaMacchia, and E. Simon. XML-signature syntax and processing, Feb. 2002.

[4] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, and V. Josifovski. Streaming XPath Processing with Forward and Backward Axes. In *Proceedings of the 19th International Conference on Data Engineering (ICDE03)*, 2003.

[5] A. Benameur, F. A. Kadir, and S. Fenet. XML Rewriting Attacks: Existing Solutions and their Limitations. In *IADIS Applied Computing 2008*. IADIS Press, Apr. 2008.

[6] Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Simon. XML path language (XPath), version 2.0, Jan. 2007.

[7] K. Bhargavan, C. Fournet, and A. D. Gordon. Verifying policy-based security for web services. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 268–277, 2004.

[8] J. Boyer, M. Hughes, and J. Reagle. XML-signature xpath filter 2.0, Nov. 2002.

[9] J. Callas, L. Donnerhacke, H. Finney, and R. Thayer. OpenPGP message format, Nov. 1998.

[10] A. Ekelhart, S. Fenz, G. Goluch, M. Steinkellner, and E. Weippl. Xml security - a comparative literature review. *J. Syst. Softw.*, 81(10):1715–1724, 2008.

[11] S. Gajek, L. Liao, and J. Schwenk. Breaking and fixing the inline approach. In *SWS '07: Proceedings of the 2007 ACM workshop on Secure web services*, pages 37–43, New York, NY, USA, 2007. ACM.

[12] G. Gou and R. Chirkova. Efficient algorithms for evaluating xpath over streams. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 269–280, New York, NY, USA, 2007. ACM.

[13] B. Kaliski. PKCS#7: Cryptographic message syntax standard, version 1.5, Mar. 1998.

[14] M. McIntosh and P. Austel. XML signature element wrapping attacks and countermeasures. In *Workshop on Secure Web Services*, 2005.

[15] M. McIntosh, M. Gudgin, K. S. Morrison, and A. Barbir. Basic security profile version 1.0. *Web Services Interoperability Organization Deliverables*, 2007.

[16] A. Nadalin, C. Kaler, R. Monzillo, and P. Hallam-Baker. Web Services Security: SOAP Message Security 1.1 (WS-Security 2004). *OASIS Standard*, 2006.

[17] D. Olteanu, T. Furche, and F. Bry. An efficient single-pass query evaluator for xml data streams. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 627–631, New York, NY, USA, 2004. ACM.

[18] M. A. Rahaman, R. Marten, and A. Schaad. An inline approach for secure soap requests and early validation. OWASP AppSec Europe, 2006.

[19] M. A. Rahaman and A. Schaad. Soap-based secure conversation and collaboration. In *ICWS*, pages 471–480, 2007.

[20] M. A. Rahaman, A. Schaad, and M. Rits. Towards secure soap message exchange in a soa. In *Workshop on Secure Web Services*, 2006.