An Overview and Evaluation of Web Services Security Performance Optimizations

Robert A. van Engelen and Wei Zhang
Department of Computer Science, Florida State University, Tallahassee, FL32306
{*engelen,wzhang*}*@cs.fsu.edu*

Abstract

WS-Security is an essential component of the Web services protocol stack. WS-Security provides end-to-end security properties (integrity, confidentiality, and authentication) through open XML standards. End-to-end message security assures the participation of non-secure transport intermediaries in message exchanges, which is a key advantage for Web-based systems and service-oriented architectures. However, point-to-point message security based on TLS (Transport Layer Security) is known to significantly outperform WS-Security. In this paper we analyze the overhead of the WS-Security protocol processing stages and evaluate existing and new techniques for WS-Security signature performance optimizations to speed up end-to-end message integrity assurance and authentication.

1. Introduction

Web Services Security (WS-Security) [11] is an essential component of the Web services protocol stack to provide end-to-end integrity, confidentiality, and authentication capabilities to Web services. End-to-end message security assures the participation of non-secure transport intermediaries in message exchanges, which is a key advantage for Web systems and service-oriented architectures.

WS-Security is based on open W3C-approved XML standards, XML Encryption [22] and XML Signature [23]. These standards are platform neutral, thus promoting interoperability. As a tradeoff, WS-Security introduces significant overhead to SOAP/XML-based messaging due to the inherent cost of cryptographic operations on XML messages and separate message parts requiring XML document parsing, (re)formatting, and conversion [3, 5, 6].

By contrast, point-to-point security protocols, such as TLS (Transport Layer Security) [10, 21], are fast but provide limited security options for Web services. While the point-to-point limitation is not always detrimental for simple services architectures requiring secure messaging, e.g. when two parties communicate over HTTPS, securing service-oriented architectures involving many intermediary processing nodes poses the problem of an untrusted "man-in-the-middle" node. All intermediary nodes must be trusted to work with TLS, requiring more elaborate key exchange mechanisms. Furthermore, TLS requires negotiation via handshake, thereby significantly complicating one-way message exchange patterns and connectionless message exchanges. For example, Web services may be bound to non-HTTP transport layer protocols such as FTP and SMTP.

Several authors conducted performance evaluation studies of WS-Security. In [3] and also [6] the authors compare the performance of WS-Security operations and choices of signature and encryption algorithms to non-secure messages using various message sizes and complexities. In [5] the authors compare the performance of Web Services, WS-Security, RMI and EMI-SSL extensively. Also in [14] a comparison is made of security mechanisms based on WS-Security, specifically for Grid services. These studies conclude that WS-Security message processing is slow, as much as a factor of 100 slower compared to non-secured SOAP/XML messaging.

It should be noted that the previously-published results assumed worst-case scenarios. Furthermore, the prior work does not consider combinations of WS-Security optimizations. In this paper we analyze, compare, and combine optimization techniques. We also present a new message digest-based caching strategy to further improve WS-Security performance. Thus, we study the performance impact of existing and new optimizations in various combinations to gain a better understanding of the performance issues and the overall impact of optimizations. Our experimental results show that specific combinations of optimizations can achieve a four-fold increase in speed for WS-Security signatures algorithms. However, these results also show that TLS remains an order of magnitude more efficient compared to the best WS-Security optimization.

978-0-7695-3310-0/08 $25.00 © 2008 IEEE
DOI 10.1109/ICWS.2008.102

The remainder of this paper is organized as follows. Section 2 gives an overview of existing techniques for WS-Security performance optimization and presents a message digest-based caching strategy to further improve the performance of WS-Security message integrity processing. A performance evaluation of the optimizations is given in Section 3. Section 4 summarizes our findings.

2. An Overview of WS-Security Optimizations

This section describes implementation techniques and practical usage strategies to optimize WS-Security. Because there is currently no published XML encryption optimization for WS-Security, the main focus of the following is on signature processing. Some of the optimizations are applicable to encryption, such as the choice of security tokens.

2.1. The Choice of Security Tokens

In [9] the authors study the impact of the Kerberos token profile impact on WS-Security performance. The Kerberos token profile method increases the packet throughput by 28% compared to the PKI X.509 token profile under full CPU load. While this is impressive, WS-Security message encryption and signing with HMAC keys can be an order of magnitude faster compared to RSA/DSA keys, assuming the X.509 token profile is used. Significant performance gains can be realized by using keys that incur low overhead, such as HMAC symmetric keys. The performance impact of HMAC versus DSA and RSA algorithms is analyzed and discussed in Section 3.

The obvious disadvantage of HMAC is that symmetric keys require the establishment of a shared secret. To address this problem, well-established key exchange methods should be used to provide a secure mechanism for mutually agreeing on a shared secret key. At the transport level the shared secret key is usually ephemeral, i.e. established for the duration of message encryption in point-to-point TLS [10, 21]. However, in WS-Security the use of a (ephemeral) shared secret key token is not straight forward, since there is no handshake protocol in WS-Security. The WS-SecureConversation [12] specification defines extensions to allow security context establishment and sharing, and session key derivation, e.g. HMAC keys. WS-SecureConversation relies on WS-Trust [13]. Some schemes use a hashed password of the sender's credentials as the HMAC key assuming that messages are both authenticated and signed.

2.2. Digest-Based Caching Strategies

In [1,15] the authors describe a caching strategy referred to as "Differential Deserialization" (DDS) to speed up the deserialization phase in Web services. The DDS technique exploits similarities between incoming SOAP/XML messages to limit deserialization to objects/values that are different within a single message and/or across multiple messages. This is accomplished by comparing checksums of parts of the inbound message against a cache of previously deserialized objects and their checksums, or by checking the state transition path of an automaton created by a previous deserialization pass. When a checksum or an automaton's state transition path matches, the deserialized object is retrieved. Thus, an XML message that is identical to a previous message does not require deserialization.

Algorithm 1: Digest-based object caching

```
Input: XML DS-signed XML message
Output: Deserialized objects or signed XML processing error
begin
    Verify ds:SignatureValue of ds:SignedInfo with the corresponding
    HMAC key, or DSA/RSA public keys;
    if signature is valid then
        foreach ds:Reference containing a URI, digest method DM,
        digest value DV, and URI-referenced element E in the message
        do
            if URI is a UUID and DM=SHA1 then
                Search ID=URI in data store or cache;
                if ID found with record (ID,HA,Object) and DV=HA
                then
                    Replace E with deserialized Object;
                else
                    Compute HA=SHA1(E);
                    if DV≠HA then
                        fail();
                    else
                        Object = deserialize(E);
                        Add (ID,HA,Object) to data store/cache;
                    end
                end
            else
                Compute HA=DM(E);
                if DV≠HA then
                    fail();
                else
                    Object = deserialize(E);
                end
            end
        end
    else
        fail();
    end
end
```

In the same spirit as DDS, we introduce a digest-based caching strategy to optimize WS-Security. The digest-based caching strategy exploits the hash values of signed XML objects by the WS-Security signature algorithm. The signature algorithm verifies the signature of the set of hash values of signed XML objects. Because the hash value is unchanged for identical XML objects, previously parsed and deserialized XML objects can be retrieved from a cache indexed by digest hash value.

Our digest-based caching algorithm is shown in Algorithm 1. A cache is used to store previously deserialized objects. The cache is indexed by digest hash value (e.g. us-

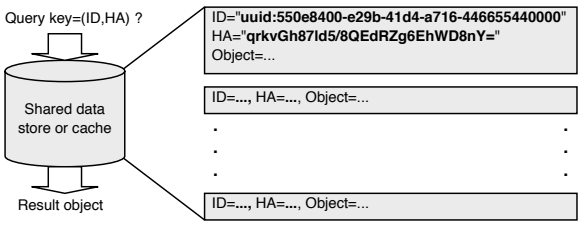


Figure 1. Digest-based object store.

ing SHA1) and UUID, see Figure 1. Upon parsing an inbound message, the algorithm first extracts the IDs and digest hash values of XML elements from the WS-Security signature. The corresponding deserialized object is then retrieved from the cache if the UUID and hash value match (a cache hit), otherwise the object is deserialized and put in the cache (a cache miss) thereby possibly evicting another object under an LRU policy. The algorithm continuously applies a cycle of cache lookup, retrieval, and update for all signed XML elements over all inbound and digitally signed SOAP/XML messages.

Our digest-based caching scheme improves the performance for both parsing and deserialization of digitally signed XML messages. In contrast to DDS, also the parsing effort is reduced, because parsing is not needed when the signature's hash value matches and the object can be immediately retrieved from cache. The additional time saved by eliminating the cost of parsing and rehashing is significant, because the ratio of XML parsing to deserialization is high (as high as 50% as reported in [19]).

At first glance it may not be obvious that the digest-based caching algorithm provides protection against message tampering. However, note that only objects with valid hash values are stored in cache or store, since the receiver always recalculates the hash values before committing the objects to the store. When the UUID and the hash values match an object in the cache, the object is retrieved and used. The referenced XML element can be safely discarded.

The chance of a cache entry collision caused by two different objects with the same hash value is extremely unlikely with SHA1. Suppose that n objects taken from $> n$ messages share the same ID, then the probability that two objects map to the same cache entry is $p(n) \approx 1 - e^{-n^2/2k}$, with $k = 2^{160}$ for SHA1. Then, $p > 10^{-9}$ (one in a billion chance) for $n > 10^{17}$. There is no computer with sufficient memory to hold that many objects ($n = 10^{17}$), that is, collisions are extremely unlikely. A cache entry collision only affects performance. Correctness is assured by verification of both the hash value of the object and UUID.

A major disadvantage of caching strategies is that the performance benefit comes at a cost of potentially significant memory usage. Memory consumption not only depends on the size of SOAP/XML message, it also depends on the variability of values in the message exchanges. Message exchanges of repeated values do not require significant memory consumption, since only a few values are stored in the cache. However, if the XML message content varies significantly, then the cache has to be large to hold the content for future comparisons resulting in huge memory resource needs.

2.3. Streaming versus Buffering Strategies

WS-Security is typically implemented based on DOM construction and manipulation. Constructing and traversing a DOM tree is inefficient both in terms of processing efficiency and memory usage. This is especially problematic for large SOAP/XML messages processed by resource-limited devices.

In [8] and [17] streaming techniques for sending XML messages are described, including stream-based WS-Security implementations that avoid DOM construction and associated character code conversations to improve the performance of WS-security. However, WS-Security requires hashing of the SOAP Body content prior to sending, since the digest hash-based signature is put in the SOAP Header that precedes the SOAP Body part. Therefore, two passes over the SOAP Body are needed (hashing and sending). This can be expensive when each pass requires serialization.

In our evaluation results, we tested the performance of WS-Security using the gSOAP [16] toolkit's implementation of WS-Security optimizations with streaming and selective buffering. The results are discussed in Section 3. On the one hand, message buffering may improve performance when message serialization is expensive, because streaming requires an additional serialization pass. On the other hand, buffering incurs memory overhead, though the memory overhead is limited to XML message size in text format.

2.4. Prehashing Strategies

In [2] the authors present a technique to speed up SOAP/XML message serialization, referred to as "Differential Serialization" (DS). The basic idea is to serialize only the differences, thus previously serialized objects are stored in XML form in a cache to resend. The same technique can be used to optimize WS-Security signatures by prehashing the signed XML elements of serialized objects. We made an important additional tweak. For each signed element in the outbound message, the digest hash value is stored with the serialized XML content in the cache. This effectively prehashes the serialized objects. Therefore, re-serialization and rehashing of the object in subsequent message exchanges are not required, thereby substantially reducing the overhead of sending WS-Security messages.

This optimization requires caching of the serialized content of all signed elements in a message body, even when reuse is low in subsequent messages. When reuse is low, the overhead of the prehashing scheme and memory usage may outweigh the savings achieved.

2.5. C14N and Re-canonicalization

The exclusive XML canonicalization (C14N-exc) standard defines a set of formatting rules to produce canonical XML. The use of canonicalized XML in the WS-Security signature process avoids problems caused by the inherent flexibility of XML processing. XML allows documents to be changed in various ways and still be considered equivalent.

In [7] the authors describe a streaming validation model for signature processing to reduce the cost of re-canonicalization at the receiving side. They point out that re-canonicalization is a substantial part of the overhead. However, optimized canonicalization implementations [17] reduce the overhead to a fraction of the total cost.

Nevertheless, canonicalization overhead can be eliminated by assuming that re-canonicalization is not necessary in most cases. The objective is to re-canonicalize when the signature verification fails due to non-canonicalized content. To implement on-demand re-canonicalization, the receiver reprocesses the entire message after a signature verification mismatch while applying C14N-exc rules. This method reduces overhead as long as most messages do not need re-canonicalization.

On-demand re-canonicalization does not guarantee speedups. Suppose a fraction $0 \leq \alpha \leq 1$ of the messages do not require re-canonicalization, because the sender generated canonical XML is received in unmodified form (e.g. when processed through communication intermediaries). Let

$$cost_{\mathrm{C14N}} = \frac{\text{re-canonicalization time}}{\text{total time}} \quad (1)$$

denote the fraction of C14N-exc re-canonicalization overhead. Assuming a unit cost of processing a message with re-canonicalization, the expected message processing cost for the optimized scheme is given by

$$cost = \alpha(1 - cost_{\mathrm{C14N}}) + (1 - \alpha)(2 - cost_{\mathrm{C14N}}) \quad (2)$$

The second term aggregates the cost of a failed verification without C14N-exc re-canonicalization ($cost = 1 - cost_{\mathrm{C14N}}$) plus the total cost of reprocessing the message ($cost = 1$) with re-canonicalization.

To gain a performance benefit we need $cost \leq 1$, thus $\alpha \geq 1 - cost_{C14N}$. Therefore, the lower the re-canonicalization cost the higher the success rate α should be to gain any performance increases. For example, in our experiments we found $cost_{C14N} = 0.12$, which requires $\alpha \geq 0.88$. Hence, if 88% of the WS-Security messages do not require re-canonicalization then performance will be increased.

To save time at the sending side, the sender could disable canonicalization in the signature processor. As a consequence, this also reduces the need for the receiver to re-canonicalize the XML. However, interoperability can no longer be guaranteed because intermediary processing nodes could still change the XML layout and thereby invalidate the signature.

3. Performance Evaluation Results

In this section we present our performance evaluation of the WS-Security optimization techniques discussed in the previous section.

3.1. Experimental Setup

We measured the round trip time of SOAP/XML request-response messages with and without WS-Security over HTTP and HTTPS (using OpenSSL TLSv1). To obtain results expressed in CPU time, the TCP/IP and socket overhead was eliminated from the presented performance timings by using memory-based messages, unless specifically indicated otherwise. Thus, network latencies are not included in the CPU tests.

All measurements were conducted using a highly optimized and tightly integrated C-based implementation in a modified version of gSOAP 2.7.9 [16, 18] with OpenSSL 0.9.7l. Performance comparisons [4,19,20] have shown that the gSOAP has very fast parsers and deserializers. This is important, because WS-Security algorithm overhead dominates overall performance rather than XML handling.

The messages used in the experiments are EchoString and EchoStringArray SOAP RPC-encoded arrays containing repeated string elements as defined per Whitemesa's SOAP 1.1 RPC round 2 interoperability [24]. Performance was measured on a 2.33 GHz Intel Core 2 Duo with 2GB 667 MHz DDR2 SDRAM using GCC 4.0.1 -O2 on a MAC OS 10.4.11 operating system. Experiments were repeated 100 to 10000 times, discarding the first iteration (warm up, stabilization), and by taking the average CPU time.

3.2. WS-Security Operations Breakdown

The choice of cryptographic algorithms and associated token keys plays a critical role in the performance of WS-Security operations, see e.g. [5, 8, 9]. Figure 2 shows the performance WS-Security authentication with symmetric HMAC key and with DSA/RSA public key. The results indicate that the performance of HMAC is an order of mag-

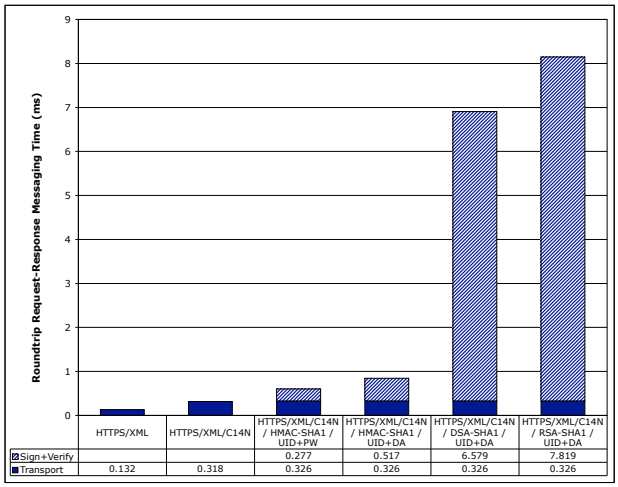


Figure 2. Total time (ms) of canonical (C14N) SOAP/XML request-response messaging over HTTPS with and without HMAC-SHA1, DSA-SHA1, and RSA-SHA1 signatures and UID authentication with passwords (PW) or password-digest authentication (DA).

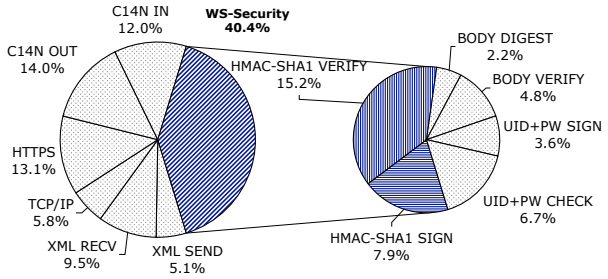


Figure 3. Timing breakdown of HMAC-based WS-Security operations (% of total time).

nitude better than that of DSA/RSA. Hence, any optimization with combination of asymmetric public key has limited effect on overall performance. The results also indicate that HTTPS with TLSv1 is much more efficient than WS-Security, even for HMAC.

To analyze WS-Security overhead in combination of HMAC, Figure 3 shows the timing breakdown of operations as percentage of the total time. The timing breakdown is similar for DSA/RSA token keys, except for the RSA/DSA-SHA1 SIGN and RSA/DSA-SHA1 VERIFY parts that completely dominate the total time of WS-Security signature with RSA or DSA.

From Figure 3, we can conclude that WS-Security algorithms with fast HMAC (40.4%) and the required XML manipulations (C14N 12.0% and 14.0%) are dominating the messaging overhead (66.4%). An explanation of these operations is given in Table 1.

Operation	Description
XML SEND	XML serialization and SOAP composition
XML RECV	XML parsing and deserialization
TCP/IP	BSD socket operations (send & recv)
HTTPS	TLSv1 operations (send & recv)
C14N OUT	XML canonicalization for SHA1(XML)
C14N IN	XML re-canonicalization for SHA1(XML)
BODY DIGEST	Sign SHA1(Body)
BODY VERIFY	Verify SHA1(Body)
UID+PW SIGN	Sign SHA1(UID,PW)
UID+PW CHECK	Verify SHA1(UID,PW) and check PW
UID+DA SIGN	DA=SHA1(UDI,PW,nonce,timestamp) and sign SHA1(UID,DA,nonce,timestamp)
UID+DA CHECK	DA=SHA1(UDI,PW,nonce,timestamp), verify SHA1(UID,DA,nonce,timestamp), check DA and UID in database
HMAC-SHA1 SIGN	Sign SHA1(SignedInfo) using K_{HMAC}
HMAC-SHA1 VERIFY	Verify SHA1(SignedInfo) using K_{HMAC}
DSA-SHA1 SIGN	Sign SHA1(SignedInfo) using K_{DSApriv}
DSA-SHA1 VERIFY	Verify SHA1(SignedInfo) using K_{DSApub}
RSA-SHA1 SIGN	Sign SHA1(SignedInfo) using K_{RSApriv}
RSA-SHA1 VERIFY	Verify SHA1(SignedInfo) using K_{RSApub}

Table 1. WS-Security messaging operations.

These results show that HMAC-SHA1 verification of the signature is slower than signing and it is the most expensive individual operation, taking 15.2% of the total time. The results also indicate that single SOAP Body element signing (BODY DIGEST) and verification (BODY VERIFY) are reasonably fast. The former contributes 2.2% overhead while the later introduces 4.8%. Both signing and verification of message content typically involve finding and locating wsu:Id attributed elements and calculating hash values for the elements as part of the signing and verification process. Searching the wsu:ID attributed elements in an inbound message for signature verification is expensive (4.8% versus 2.2%). XML re-canonicalization (C14N IN) also contributes to the messaging overhead by 12.0%. The XML canonicalization output operation (C14N OUT) took 14.0% of the total time.

From Figure 3 we can also conclude that UID+DA digest authentication can be expensive. It took 2.7ms on average to check digest authentication and protect against replay attacks using a container with 10,000 distinct UID and nonce (*UID, nonce*) pairs.

3.3. Performance of Digest-Based Caching

From the results shown in Figure 3 we see that HMAC-SHA1 VERIFY, C14N IN, and XML RECV take 36.7% of the total time. The digest-based caching strategy discussed in Section 2.2 leverages this opportunity to optimize performance by reducing these costly computations and deserialization by reusing already deserialized objects based on their WS-Security signature digest values.

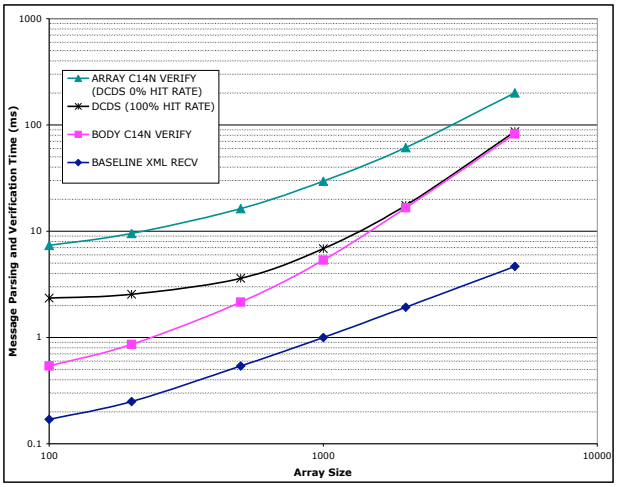


Figure 4. Performance of message parsing with HMAC-SHA1 signature verification for various message sizes (array size).

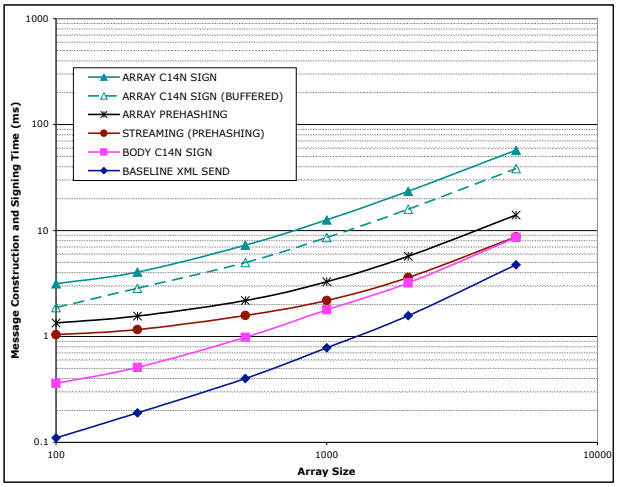


Figure 5. Performance of message construction with HMAC-SHA1 signatures for various message sizes (array size).

Figure 4 shows the performance of the digest-based caching algorithm for EchoStringArray with the HMAC-SHA1 signature. The performance of the algorithm is compared to the parsing, deserialization, and verification of each array element individually without the digest-based caching optimization (ARRAY C14N VERIFY), and the the parsing, deserialization, and verification of a single signature (BODY C14N VERIFY), i.e. the signature of the SOAP Body element that contains the entire array. For comparison, the performance of the baseline non-signed message parsing and deserialization (BASELINE XML RECV) is shown also in the figure.

The results indicate that digest-based caching optimization can improve performance up to a factor of 3 or 4. Note that the ARRAY C14N VERIFY performance is identical to the performance of digest-based caching with 0% hit rate (all miss). Based on this it is reasonable to conclude that performance of digest-based caching for WS-Security is more than twice faster on average than that of the non-optimized version.

However, single root element verification such as on the SOAP Body still gives best performance, at the cost of reduced flexibility of signing individual elements. Digest-based caching verification performs as good as single body verification only in the limit, i.e. for large message sizes. Therefore, digest-based caching (and related differential deserialization methods [1, 15]) should only be used as an optimization when the hit ratio is close to optimal and a large number of elements is signed.

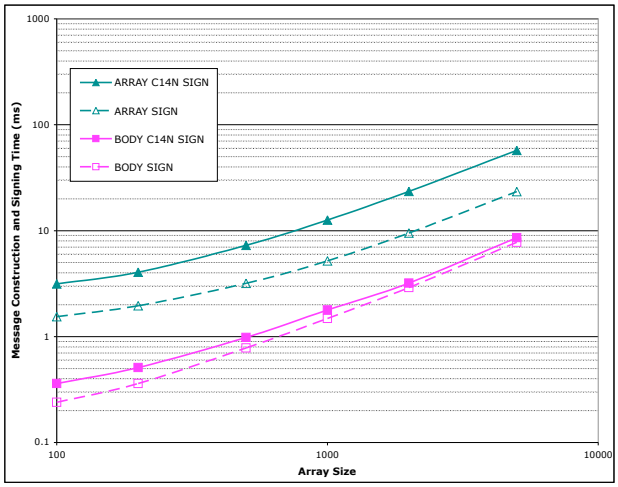
3.4. Performance of Streaming and Prehashing

From the results shown in Figure 3 we see that HMAC-SHA1 SIGN, C14N OUT, and XML SEND take 27% of the total time. Streaming with prehashing reduces the overhead of these processing parts to optimize the WS-Security performance. Assuming that the prehashed elements are stored in canonicalized form, the C14N OUT (14%) time can be eliminated.

Figure 5 shows the performance of streaming with prehashing (Section 2.4) of messages with arrays of 100 to 5,000 elements (STREAMING PREHASHING), where each array element is individually signed. In the streaming method each array element was prehashed prior to message construction and signing with HMAC-SHA1. The speed improvement of streaming compared to two-pass message construction and signing (ARRAY C14N SIGN) ranges from a factor 3 (for 100 array elements) to 7 (for 5,000 array elements). Prehashing requires two passes (ARRAY PREHASHING). From the results it can be concluded that eliminating a pass by streaming (STREAMING PREHASHING) reduces overhead by another 20% to 35%.

When prehashing is not used, streaming alone may not improve performance, because messages have to be emitted twice in two passes to compute the digest hash first (ARRAY C14N SIGN), see Section 2.3). In this case buffering improves performance at a cost of increased memory use (ARRAY C14N SIGN BUFFERED).

The results shown in Figure 5 indicate that the speed of streaming large messages with a high number of signed el-



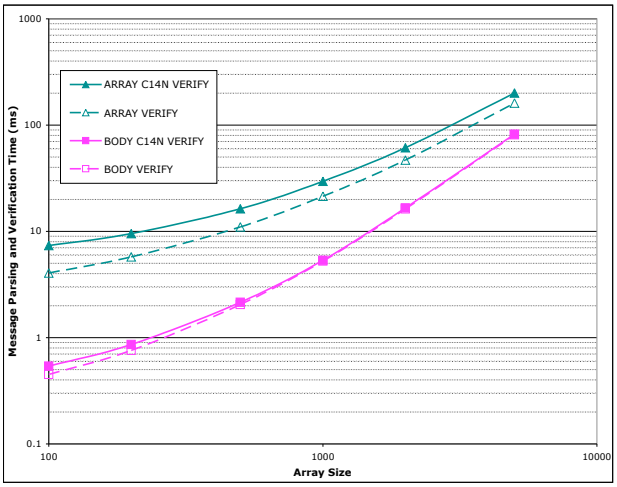


Figure 6. Message canonicalization overhead of sends (left) and receives (right) of single SOAP Body signatures with HMAC-SHA1 versus multiple signed array elements for increasing array size.

ements converges to the speed of a message with a single signed element (BODY C14N SIGN). Signing a single root element such as the SOAP Body gives the best performance, but at the cost of reduced flexibility. By individually signing elements, the elements can be isolated for further processing by a segmented SOAP stack.

Overall, the results show that the WS-Security HMAC-SHA1 signature process is costly and slows messaging down by a factor of 2 to 3 compared to the baseline performance. On the other hand, the signature process scales well with increasing message size.

3.5. Performance Impact of C14N

Figure 6 shows the C14N-exc re-canonicalization overhead (C14N) for constructing HMAC-SHA1-signed messages and parsing and verifying EchoStringArray inbound messages. The re-canonicalization overhead of a single signed part (the SOAP Body containing an array of elements) versus multiple signed parts (each array element is individually signed) is shown for comparison. From the figure it can be concluded that the canonicalization overhead is small for a single signed part, such as the SOAP Body, but can be significant for many signed parts, such as the individually signed elements of an array.

Figure 6 (left) shows the C14N-exc overhead for message construction and HMAC-SHA1 signing. For a single-signed SOAP Body the overhead varies from 0.1 ms (or 50% overhead) to 0.8 ms (or 1.0% overhead) for increasing message body size from 10KB (100 array elements) to 230KB (5,000 array elements), respectively. For multiple signed elements, the relative canonicalization overhead on the sending side grows with increasing number of signed array elements in the message, from 1.6 ms (or 50% overhead) to 34 ms (or 59% overhead) for increasing message body size from 88KB (100 signed array elements) to 1,066KB (5,000 signed array elements), respectively.

Figure 6 (right) shows the C14N-exc re-canonicalization overhead on inbound message parsing and verification. For a single-signed SOAP Body the overhead varies from 0.1 ms (or 19% overhead) to 0.9 ms (or 1.0% overhead) for increasing message body size from 10KB (100 array elements) to 230KB (5,000 array elements), respectively. For multiple signed elements, the relative canonicalization overhead on the receiving side shrinks with increasing number of signed array elements in the message, from 3.3 ms (or 45% overhead) to 40 ms (or 20% overhead) for increasing message body size from 88KB (100 signed array elements) to 1,066KB (5,000 signed array elements), respectively.

From these results it can be concluded that if a sender's and receiver's C14N-exc policies can be matched, the overall performance can be increased by avoiding re-canonicalization at the sending and/or receiving side. Note that if the match ratio α in Eq. (2) is high, then reprocessing the message upon signature failure is a viable option as explained in Section 2.5. In this way performance can be improved up to 10% to 20% for message with multiple elements signing and verification.

4. Conclusion

It is well known that WS-Security introduces significant overhead to XML Web service message processing due to the inherent cost of the cryptographic operations. In this

paper we compared a number of optimizations presented in the literature, introduced further improvements of these, and evaluated the performance impact of these optimizations with various optimization combinations for small to large XML messages. Significant performance differences are observed in our experiments, which can lead to an order of magnitude speed up for WS-Security uses in performance-critical Web service applications. All optimizations scale well with message size, thus providing a predictable set of tools to improve WS-Security performance. Differential deserialization-based methods [1, 15] are not effective to reduce the cryptographic cost of message integrity verification, due to the additional processing overhead of hashing many XML elements and the memory overhead for maintaining the cache. By contrast, selecting a more optimal security token key mechanism such as HMAC-SHA1, the use of message streaming techniques, exploiting prehashing, and on-demand (re)-canonicalization optimizations appear to be most effective.

References

[1] N. Abu-Ghazaleh and M. Lewis. Differential deserialization for optimized SOAP performance. In *proceedings of the ACM/IEEE conference on Supercomputing*, pages 21–31, Los Alamitos, CA, 2005. IEEE Computer Society.

[2] N. Abu-Ghazaleh, M. Lewis, and M. Govindaraju. Differential serialization for optimized SOAP performance. In *proceedings of the IEEE International Symposium on High Performance Distributed Computing*, pages 55–64, Los Alamitos, CA, 2004. IEEE Computer Society.

[3] S. Chen, J. Zic, K. Tang, and D. Levy. Performance evaluation and modeling of Web services security. In *Proceedings of the IEEE International Conference on Web Services (ICWS'07)*, pages 431–438, 2007.

[4] M. R. Head, M. Govindaraju, R. van Engelen, and W. Zhang. Benchmarking XML processors for applications in Grid Web services. In *proceedings of ACM/IEEE Supercomputing Conference*, Los Alamitos, CA, November 2006. IEEE Computer Society.

[5] M. B. Juric, I. Rozman, B. Brumen, M. Colnaric, and M. Hericko. Comparison of performance of Web services, WS-Security, RMI, and RMI-SSL. *Journal of Systems and Software*, 79(5):689–700, 2006.

[6] H. Liu, S. Pallickara, and G. Fox. Performance of Web services security. In *13th Annual Mardi Gras Conference*, Baton Rouge, Lousiana, USA, Feburay 2005.

[7] W. Lu, K. Chiu, A. Slominski, and D. Gannon. A streaming validation model for SOAP digital signature. In *In 14th IEEE International Symposium on High Performance Distributed Computing (HPDC-14)*, 2005.

[8] S. Makino, K. Tamura, T. Imamura, and Y. Nakamura. Implementation and performance of WS-Security. *Int. J. Web Service Res.*, 1(1):58–72, 2004.

[9] A. Moralis, V. Pouli, M. Grammatikou, S. Papavassiliou, and V. Maglaris. Performance comparison of Web services security: Kerberos token profile against X.509 token profile. In *ICNS '07: Proceedings of the Third International Conference on Networking and Services*, page 28, Washington, DC, USA, 2007. IEEE Computer Society.

[10] Network Working Group. The Transport Layer Protocol Version 1.1 (RFC4346). Available at http://www.faqs.org/rfcs/rfc4346.html.

[11] Oasis Consortium. WS-Security specification, 2004. Available from www.oasis-open.org.

[12] Oasis Consortium. Web services secure conversation language (WS-SecureConversation), 2005. Available from www.oasis-open.org.

[13] Oasis Consortium. Web Services Trust Language (WS-Trust), 2005. Available from www.oasis-open.org.

[14] S. Shirasuna, A. Slominski, L. Fang, and D. Gannon. Performance comparison of security mechanisms for Grid services. In *GRID '04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*, pages 360–364, Washington, DC, USA, 2004. IEEE Computer Society.

[15] T. Suzumura, T. Takase, and M. Tatsubori. Optimizing Web services performance by differential deserialization. In *ICWS '05: Proceedings of the IEEE International Conference on Web Services (ICWS'05)*, pages 185–192, Washington, DC, USA, 2005. IEEE Computer Society.

[16] R. van Engelen. The gSOAP toolkit for C and C++ Web services, 2001. Available from http://gsoap2.sourceforge.net.

[17] R. van Engelen. A framework for service-oriented computing with reusable C and C++ Web service components. *accepted for publication in ACM Transactions on Internet Technologies*, 2008.

[18] R. van Engelen and K. Gallivan. The gSOAP toolkit for Web services and peer-to-peer computing networks. In *proceedings of the IEEE International Symposium on Cluster Computing and the Grid*, pages 128–135, Los Alamitos, CA, May 2002. IEEE Computer Society.

[19] R. A. van Engelen. Constructing finite state automata for high performance XML Web services. In *proceedings of the International Symposium on Web Services (ISWS)*, Las Vegas, NV, 2004. CSREA Press.

[20] R. A. van Engelen. A framework for service-oriented computing with C and C++ Web service components. *To appear in ACM Transactions on Internet Technologies*, 2008.

[21] J. Viega, M. Messier, and P. Chandra. *Network Security with OpenSSL*. O'Reilly, 2002.

[22] W3 Consortium. XML Encryption specification. Available from www.w3.org.

[23] W3 Consortium. XML Signature specification. http://www.w3.org/TR/xmldsig-core/.

[24] White Mesa. White Mesa Interop Lab. Available from www.whitemesa.com/interop.htm.