One Bad Apple: Backwards Compatibility Attacks on State-of-the-Art Cryptography

Tibor Jager
Horst Görtz Institute for IT Security
Ruhr-University Bochum
tibor.jager@rub.de

Kenneth G. Paterson*
Information Security Group
Royal Holloway, University of London
kenny.paterson@rhul.ac.uk

Juraj Somorovsky†
Horst Görtz Institute for IT Security
Ruhr-University Bochum
juraj.somorovsky@rub.de

Abstract

Backwards compatibility attacks are based on the common practical scenario that a cryptographic standard offers a choice between several algorithms to perform the same cryptographic task. This often includes secure state-of-the-art cryptosystems, as well as insecure legacy cryptosystems with known vulnerabilities that are made available for backwards compatibility reasons.

Obviously using insecure legacy cryptosystems is dangerous. However, we show the less obvious fact that even if users have the best of intentions to use only the most up-to-date, vulnerability-free version of a system, the mere existence of support for old versions can have a catastrophic effect on security.

We demonstrate the practical relevance of our results by describing attacks on current versions of important cryptographic Web standards: W3C XML Encryption and XML Signature, and JSON Web Encryption and Web Signature. We furthermore propose practical and effective countermeasures thwarting backwards compatibility attacks. These can be applied in new versions of these standards as well as in related specifications applying cryptographic primitives.

*This author was supported by EPSRC Leadership Fellowship EP/H005455/1

†This author was supported by the Sec2 project of the German Federal Ministry of Education and Research (BMBF, FKZ: 01BY1030)

1 Introduction

Complexity is often portrayed as being the enemy of security: the more complex a system is, the harder it is to analyse, and the harder it is to eliminate all possible attack vectors. One source of complexity in real world security systems stems from the desire to maintain backwards compatibility between new and old versions of systems. This may continue to be the case in spite of attacks against the old systems. In extreme cases, the legacy argument is sometimes used: a certain system cannot be switched off despite having known vulnerabilities because it runs a mission-critical application that cannot be supported in any other way.

We have seen this kind of technology development path being followed many times. For example, in the context of secure protocols, it is by now well-known that "encryption-only" configurations of IPsec are vulnerable to active attacks which recover full plaintext [12, 59, 22]. Yet these configurations are still allowed by the current third generation of IPsec RFCs and still supported by many vendors. As another example, it has been known since 1995 [62] that using chained initialization vectors (IVs) in CBC mode undermines the security of protocols like SSL/TLS, in the sense of allowing distinguishing attacks against the protocols. TLS 1.1 [23], published in 2006, removed support for chained IVs. Despite this distinguishing attack having been turned into a full plaintext recovery attack [25], TLS 1.0 still remains in widespread use.

It is obvious that introducing a new system whilst maintaining backwards compatibility with old versions having known weaknesses undermines security: if a system or a protocol *can* be configured into an insecure state, then some users *will* do so. In this paper, we show something a little

less obvious. Namely, that even if users have the best of intentions to use only the most up-to-date, vulnerability-free version of a system, the mere *existence* of support for old versions can have a catastrophic effect on security. We show this in the context of systems employing cryptography, introducing what we term *backwards compatibility (BC)* attacks. Like all good attacks, these are obvious in retrospect, but they do not seem to have been fully explored before. As we shall discuss in more detail below, they are closely related to, but distinct from, version rollback attacks [71].

As a taster of our attacks to follow, consider a situation where, for backwards compatibility reasons, a system still allows the use of CBC mode encryption, but where Galois Counter Mode (GCM) is the preferred secure encryption scheme. The reason to switch to GCM may be that the CBC mode is vulnerable to one of the several attacks that can, under certain circumstances, recover plaintext when it is encrypted in this mode – these attacks exploit the malleability of CBC-mode (i.e., an attacker is able to make meaningful changes to an encrypted plaintext by making purposeful changes to the ciphertext) in combination with the availability of an "oracle" telling the attacker whether modified plaintexts are still valid. Such oracles can, for instance, be based on error messages returned due to invalid padding (*"padding oracle attacks"* [68, 24]) or other properties of the plaintext, like malformed XML structure [38]. The detailed description of these oracles is beyond the scope of this paper – for us it is only important to know that *in certain scenarios an attacker is able to* decrypt *CBC-ciphertexts*, due to a weakness of CBC. Now what happens if users select GCM as their preferred mode? Then an attacker who can modify messages so that they are decrypted using CBC mode instead of GCM can use the *old* attack to decrypt the ciphertexts as if they were CBC encrypted. Here we assume that the *same* key is used, irrespective of the mode. Then, as we explain in detail in Section 2, this CBC decryption capability can be quickly and efficiently turned into a distinguishing attack against GCM.

This situation not purely hypothetical. As we will see, this is exactly the evolutionary path that has been followed in the XML Encryption standards. Very recently, the XML Encryption Working Group published a new XML Encryption standard draft [27] to recommend the use of GCM in preference to CBC mode in response to such an attack [38]. CBC mode is retained in the standard for backwards compatibility reasons. And the same key is used for both GCM and CBC mode. Finally, a man-in-the-middle attacker can easily manipulate XML document fields so that the use of CBC mode for decryption is indicated instead of GCM. So all the pre-conditions for our attack are met. Since CBC mode is mandatory, any state-of-the-art, standard-compliant implementation of XML Encryption will be vulnerable to this BC attack, even if all honest users exclusively stick to using GCM. We will demonstrate a practical distinguishing attack against an implementation of XML Encryption using this attack vector in Section 5.

This basic BC attack motivates the following questions, which we attempt to answer in this paper:

- Which other encryption modes (and, more generally, cryptographic schemes), can interact with one another badly in this kind of scenario?
- To what extent do deployed systems fall victim to this class of attack?
- What countermeasures are readily available?

The last question seems simplest to answer: use appropriate key separation to ensure that the same keys are not used in "weak" and "strong" cryptographic algorithms. However, this apparent simplicity is deceptive. Our experience is that developers sometimes fail to appreciate this requirement, or understand the requirement but fail to provide key separation because they do not want to even invest the small development effort needed to implement suitable key derivation algorithms. Moreover, in the context of public key cryptography, the most common data format for transporting public keys, the X.509 certificate, does not by default contain a field that limits the cryptographic algorithms in which a public key and its corresponding private key can be used. For example, a public key specified in an X.509 certificate as being an RSA encryption key could be used in either the PKCS#1 v1.5 or the PKCS#1 v2.1 (RSA-OAEP) encryption algorithms, with the former possibly being subject to Bleichenbacher-style [13] attacks. This lack of precision opens up the possibility of BC attacks in the public key setting.

For the first question, we do not attempt a systematic analysis of all the possibilities, since even the number of basic modes of operation of a block cipher precludes this. Instead, we examine some particularly attractive (from the attacker's point of view) cases in the symmetric and asymmetric settings. Specifically, we look at the interactions between CBC mode and GCM, this being particularly important in the context of XML Encryption, and between CBC mode and the AES Key Wrap algorithm. In both cases, we are able to mount a BC attack to break what should be a secure algorithm. In the public key setting, we focus on RSA encryption and signatures, showing a BC attack on RSA-OAEP when it is used in conjunction with an implementation of PKCS#1v1.5 encryption that is vulnerable to Bleichenbacher's attack [13]. We also remark that a signature forgery attack is possible under the same circumstances; here we require the same RSA key to be allowed for use in both encryption and signature algorithms, a situation promoted for instance by [31, 60].

To address the second question, we demonstrate working BC attacks against the most recent drafts of the W3C XML Encryption [28] and XML Signature [29] standards,

as well as against the current draft of JSON Web Encryption [41] and Web Signature [40]. In the secret key setting, we describe a practical BC attack that allows to break (i.e., to distinguish plaintexts of) GCM-based encryption in XML Encryption, based on a weakness of CBC. The basic idea of this generic attack is described in Section 2. Furthermore, in Section 5.2 we apply a significantly more efficient variant of this attack, which exploits specific weaknesses of XML Encryption, exemplarily to the widely-used Apache *Web Services Security for Java (WSS4J)* library. In the public-key setting, we show how the well-known attack of Bleichenbacher [13] gives rise to a BC attack that allows an attacker to decrypt ciphertexts of PKCS#1 v2.0 encryption in both XML Encryption [27] and JSON Web Encryption [41], and to forge signatures for arbitrary messages in XML Signature [29] and JSON Web Signature [40]. The attack principle is described in Section 4. We furthermore report on our experimental results, executed against the Java implementation of JSON Web Encryption and JSON Web Signature *Nimbus-JWT* [52], in Section 5.3.

1.1 Related Work

Wagner and Schneier [71] described *version rollback attacks* on Version 2.0 of the SSL protocol. Speaking generally, version rollback attacks target cryptographic protocols where cryptographic algorithms and parameters are negotiated interactively between communication partners at the beginning of a protocol execution. The attacker modifies messages exchanged in this negotiation phase, in order to lure both communication partners into using weak cryptography, such as for instance legacy export-weakened algorithms.

Backwards compatibility attacks can be seen as a variant of version rollback attacks that apply to non-interactive protocols. An essential difference is that version rollback attacks on two-party protocols can be prevented by either party, if that party simply uses *exclusively* strong state-of-the-art cryptography.[1] In contrast, in this paper we describe attacks that can not be prevented if one party is only prepared to use strong cryptography: the willingness of the other party to use weak cryptography suffices to foil security.

Kelsey *et al.* [47] describe *chosen-protocol attacks*. These consider a scenario where a victim executes a cryptographic protocol Π, and an attacker is able to trick this victim into executing an additional *maliciously designed* cryptographic protocol Π', too. This helps the attacker to break the security of Π. Clearly such attacks require a very strong attacker, and are only applicable if potential victims can be seduced into executing malicious protocols. In contrast, in typical backwards compatibility attacks, no adversarial control over the protocols executed by honest parties is needed.[2]

The attack described by Kaliski Jr. [45] assumes an attacker that is able to register new *hash function identifiers*, and can thus be seen as a special case of chosen-protocol attacks.

Gligoroski *et al.* [32] emphasise the need for key separation when using different modes of operation of a block cipher, and criticise some ISO and NIST standards for failing to make this point explicitly. However, they do not present any concrete attacks against deployed protocols, and their on-paper attacks do not seem to work as described (for example, their Attack 4 which attempts to exploit the interaction between CBC and CTR modes of operation seems to require the occurrence of a highly unlikely event in Step 3 of the attack).

Barkan *et al.* [8] showed that the key separation principle is violated in the GSM mobile telecommunications system, and exploited this in what can be seen as a BC attack on the GSM encryption mechanism: in their attack, an active attacker fools the receiver into using a weak encryption algorithm (A5/2), extracts the key by cryptanalysis, and then uses the same key to decrypt traffic protected by the stronger A5/1 algorithm. Thus the continued presence of a weak algorithm enables the enhanced security provided by a stronger algorithm to be bypassed. This is the only previous concrete example of a BC attack (of the specific type we explore in this paper) that we know of.

A cryptographic primitive with the property that different instantiations can securely share the same key is called *agile* [1]. In a sense, the attacks presented in this paper provide evidence that block-cipher modes of operation and public-key schemes are not agile, and show how this property leads to relatively efficient practical attacks on important Web standards. Another line of work, related to agility, concerns *joint security*, wherein a single asymmetric key pair is used for both signatures and encryption. An up-to-date overview of work in this area is provided in [58].

1.2 Responsible Disclosure

We informed W3C (who are responsible for the XML Encryption standard) of the attacks presented in this paper in July 2012. They have acknowledged the attack and are planning to extend the specification with security considerations addressing BC attacks. We informed the JOSE working group, which is in charge of JSON Web Encryption and JSON Web Signature, of our BC attack on RSA-OAEP and

[1] In presence of an attacker the negotiation might then fail, which reduces the version rollback attack to a denial-of-service attack.

[2] Even worse, in the examples of BC attacks described in this paper honest parties are *forced* to execute weak cryptographic algorithms, in order to remain standards-compliant.

PKCS#1 v1.5 in April 2012. Their standards are still under development at the time of writing.

We also communicated with several vendors applying XML Signature and XML Encryption. We highlight the steps they used to counter our attacks in Section 5.

2 Breaking GCM with a CBC Weakness

In this section we describe a BC attack on symmetric encryption. We show how to break the expected security of ciphertexts encrypted in Galois counter mode (GCM) by exploiting a weakness of the cipher-block chaining (CBC) mode.

This attack provides just one concrete example of a BC attack. We have chosen to describe this particular case in detail because we will show the practical applicability of exactly this attack in Section 5.2.

2.1 Preliminaries

We first describe the GCM and CBC modes of operation, and give a high-level description of known attacks on CBC.

2.1.1 Galois Counter Mode

Galois counter mode (GCM) [51] is a block-cipher mode of operation, which provides both high efficiency and strong security in the sense of authenticated encryption [9]. In particular GCM provides security against chosen-ciphertext attacks, like padding-oracle attacks [68, 61, 24, 38, 3], for instance. GCM is therefore an attractive choice for a replacement of modes of operation that are susceptible to such attacks.

For this reason, GCM was recently included in the XML Encryption [27] standard as a replacement for CBC, in response to the attack from [38]. It is also widely supported in other applications, like IPsec [70].

Description. In the sequel let us assume a block-cipher $(\mathsf{Enc}, \mathsf{Dec})$, consisting of an encryption algorithm Enc and a decryption algorithm Dec, with 128-bit block size[3] (like AES [2]). Let $m = (m^{(1)}, \ldots, m^{(n)})$ be a message consisting of n 128-bit blocks, where $n < 2^{32}$.[4] Let k be the symmetric key used for encryption and decryption. A message is encrypted with $(\mathsf{Enc}, \mathsf{Dec})$ in GCM-mode as follows (cf. Figure 1).

- A 96-bit *initialization vector* $iv \in \{0,1\}^{96}$ is chosen at random. A counter cnt is initialized to $\mathsf{cnt} := iv||0^{31}||1$, where 0^{31} denotes the string consisting of 31 0-bits.

[3]In [26] GCM is specified only for 128-bit block ciphers.

[4]This is the maximal message length of GCM, longer messages must be split and encrypted separately.

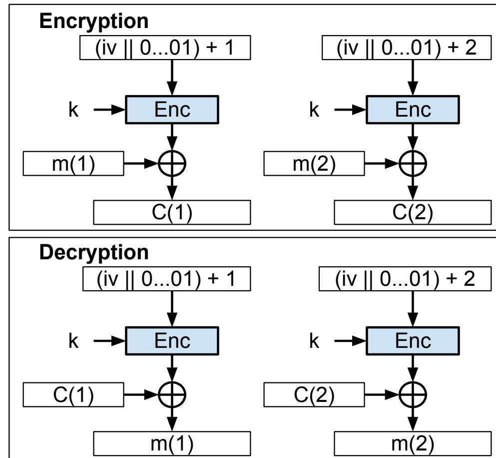

Figure 1. Counter-mode encryption and decryption, as used in Galois Counter Mode (GCM), of two message blocks.

- For $i \in \{1, \ldots, n\}$, the i-th message block[5] $m^{(i)}$ is encrypted by computing the i-th ciphertext block $C^{(i)}$ as
$$C^{(i)} := \mathsf{Enc}(k, \mathsf{cnt} + i) \oplus m^{(i)}.$$
- In parallel, an authentication tag τ (a message authentication code) is computed using arithmetic over a binary Galois field. The details of this computation are not relevant for our attack.[6]
- The resulting ciphertext is $C = (iv, C^{(1)}, \ldots, C^{(n)}, \tau)$.

The decryption procedure inverts this process in the obvious way.

2.1.2 Cipher-Block Chaining

Cipher-block chaining (CBC) [53] is presumably the most widely used block-cipher mode of operation in practice.

Let $(\mathsf{Enc}, \mathsf{Dec})$ be a block-cipher with 128-bit block size[7], let $m = (m^{(1)}, \ldots, m^{(n)})$ be a (padded) message consisting of n 128-bit blocks, and let k be the symmetric key used for encryption and decryption. A message is encrypted with $(\mathsf{Enc}, \mathsf{Dec})$ and key k in CBC-mode as follows (cf. Figure 2).

- An *initialization vector* $iv \in \{0,1\}^{128}$ is chosen at random. The first ciphertext block is computed as

$$x := m^{(1)} \oplus iv, \qquad C^{(1)} := \mathsf{Enc}(k, x). \qquad (1)$$

[5]Note that $i < 2^{32}$.

[6]In fact, the BC attack described in this section does not only apply to *Galois* counter mode, but to any "counter" mode of operation which encrypts messages in a similar way, cf. Section 3.1.

[7]CBC is specified for an arbitrary block length, we consider the special case for consistency reasons.

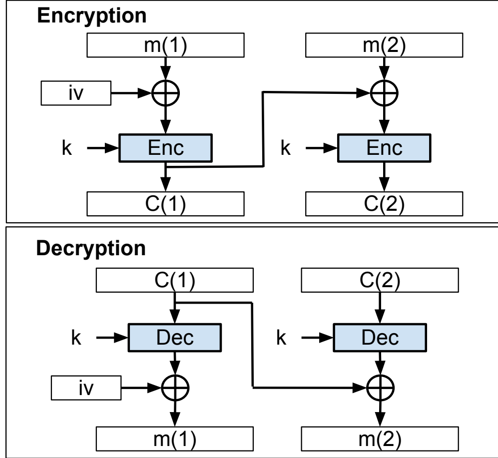

Figure 2. Cipher-block chaining (CBC) encryption and decryption of two message blocks.

- The subsequent ciphertext blocks $C^{(2)}, \ldots, C^{(d)}$ are computed as

$$x := m^{(i)} \oplus C^{(i-1)}, \qquad C^{(i)} := \mathsf{Enc}(k, x) \qquad (2)$$

  for $i = 2, \ldots, n$.
- The resulting ciphertext is $C = (iv, C^{(1)}, \ldots, C^{(n)})$.

The decryption procedure inverts this process in the obvious way.

2.1.3 Known Attacks on CBC

Starting with Vaudenay's padding-oracle attacks [68], several efficient attacks exploiting the malleability of CBC-encrypted ciphertexts have been published. Prominent targets are ASP.NET [24], XML Encryption [38], and Datagram TLS [3]. These attacks are the main reason why CBC is phased out in new standards and replaced with modes of operation like GCM that provide security against chosen-ciphertext attacks.

An abstract view on attacks on CBC. The details of these attacks will not be important for our further considerations. Only two properties that all these attacks have in common will be important for us: they allow to *decrypt* ciphertexts encrypted in CBC-mode, and they are *efficient*.

Thus, from an abstract point of view, the attacks provide an *efficient CBC decryption oracle* $\mathcal{O}_{\mathsf{CBC}}$. This oracle takes as input a CBC-encrypted ciphertext $C = (iv, C^{(1)}, \ldots, C^{(n)})$ encrypting a message $(m^{(1)}, \ldots, m^{(n)})$, and returns

$$(m^{(1)}, \ldots, m^{(n)}) = \mathcal{O}_{\mathsf{CBC}}(C)$$

2.2 The Backwards Compatibility Attack

In this section, we describe a generic backwards compatibility attack on GCM, which is based on a weakness of CBC. We will first describe an abstract application scenario, which is practically motivated by the recent development of the XML Encryption standard. Then we describe the attack that allows an attacker to determine whether a ciphertext contains a certain message, and discuss the relevance of such distinguishing attacks. Finally, we sketch optimizations of the generic attack, which lead to significant efficiency improvements.

2.2.1 Application Scenario

In the sequel let us consider a scenario (an example application) in which encrypted messages are sent from senders $S_1, \ldots, S_\ell$ to a receiver R. Each ciphertext C received by R consists of two components $C = (C_{\mathsf{pub}}, C_{\mathsf{sym}}^{\mathsf{CBC}})$, where

- C_{pub} is a public-key encryption of an ephemeral session key k under R's public-key, and
- $C_{\mathsf{sym}}^{\mathsf{CBC}}$ encrypts the actual payload data under key k, using a block-cipher in CBC-mode.

Suppose that $S_1, \ldots, S_\ell$ and R use this application, until it eventually turns out that it is susceptible to a chosen-ciphertext attack (CCA) which allows an attacker to decrypt ciphertexts in CBC-mode. For example, this may involve a padding oracle attack.

The application is immediately updated. The update replaces CBC-mode with GCM-mode, because GCM-mode provides provable CCA-security [51]. It is well-known that if the public-key encryption scheme used to encrypt the session key k is CCA-secure too,[8] then this combination forms a CCA-secure encryption scheme. Therefore senders using this combination of algorithms may expect that their data is protected against chosen-ciphertext attacks.

After the update the receiver R remains *capable* of decrypting CBC-mode ciphertexts for backwards-compatibility reasons, since it is infeasible to update the software of all senders $S_1, \ldots, S_\ell$ simultaneously. However, at least those senders that are using GCM instead of CBC may expect that their data is sufficiently protected.

We show that the latter is not true. The sole *capability* of R being able to decrypt CBC ciphertexts significantly undermines the security of GCM ciphertexts.

2.2.2 A Distinguishing Attack on GCM

We describe a *distinguishing attack*, which allows the attacker to test whether a GCM ciphertext contains a particular message.The attack exploits the CBC decryption capa-

[8] For instance, RSA-OAEP [10], standardized in RSA-PKCS#1 v2.1 [42], is a widely used public-key encryption algorithm that provably meets this security property [30].

bility of R. It can be applied block-wise to each ciphertext block, which enables the attacker to employ a "divide-and-conquer" strategy that in many scenarios is equivalent to a decryption attack. See Section 2.2.3 for further discussion of why distinguishing attacks matter.

The attack consists of two key ingredients.

1. We show that the availability of the CBC decryption attack allows the attacker not only to decrypt arbitrary ciphertexts in CBC-mode, but also to invert the block-cipher used within CBC at arbitrary positions. That is, we show that a CBC decryption oracle implies a block-cipher decryption oracle.
2. We show that this block-cipher decryption oracle can be used to mount a distinguishing attack on GCM.

CBC-Decryption implies Block-Cipher Inversion. Due to the availability of the CBC decryption attack, R involuntarily provides an efficient CBC decryption oracle $\mathcal{O}_{\mathsf{CBC}}$, which takes as input a tuple $C = (C_{\mathsf{pub}}, C_{\mathsf{sym}}^{\mathsf{CBC}})$, and returns the decryption of $C_{\mathsf{sym}}^{\mathsf{CBC}}$ under the key k contained in C_{pub}.

We show that this oracle $\mathcal{O}_{\mathsf{CBC}}$ can be turned into a new oracle $\mathcal{O}_{\mathsf{Dec}}$ that inverts the block-cipher used in CBC-mode. Oracle $\mathcal{O}_{\mathsf{Dec}}$ takes as input a tuple $C = (C_{\mathsf{pub}}, C')$, and returns the block-cipher decryption $m' = \mathsf{Dec}(k, C')$ of C' under the key k contained in C_{pub}.

Oracle $\mathcal{O}_{\mathsf{Dec}}$ proceeds on input (C_{pub}, C') as follows.

1. It chooses an arbitrary initialization vector iv'.
2. It queries the CBC decryption oracle on input

$$(C_{\mathsf{pub}}, (iv', C')).$$

   Note that (iv', C') is a valid CBC ciphertext consisting of an initialization vector iv and a single ciphertext block C'. Therefore oracle $\mathcal{O}_{\mathsf{CBC}}$ will return the CBC decryption

$$m = \mathsf{Dec}(k, C') \oplus iv$$

   of (iv', C').
3. Finally, $\mathcal{O}_{\mathsf{Dec}}$ computes and outputs $m' = m \oplus iv'$.

It is straightforward to verify that $m' = \mathsf{Dec}(k, C')$.

Distinguishing GCM Ciphertexts. Consider an attacker that eavesdrops an encrypted message $C = (C_{\mathsf{pub}}, C_{\mathsf{sym}}^{\mathsf{GCM}})$ sent from a sender S to receiver R. Ciphertext C_{pub} encrypts a key k, and $C_{\mathsf{sym}}^{\mathsf{GCM}} = (iv, C^{(1)}, \ldots, C^{(n)}, \tau)$ encrypts a message $m = (m^{(1)}, \ldots, m^{(n)})$ in GCM-mode with key k.

Assume the attacker has access to an oracle $\mathcal{O}_{\mathsf{Dec}}$ which takes as input a tuple $C = (C_{\mathsf{pub}}, C')$ where C' is a single ciphertext block, and returns the block cipher decryption of C' under the key k contained in C_{pub}.

The attacker can use this oracle to test whether the i-th encrypted message block $m^{(i)}$ contained in the eavesdropped ciphertext block $C^{(i)}$ is equal to a certain message m'. It proceeds as follows.

1. The attacker queries oracle $\mathcal{O}_{\mathsf{Dec}}$ by submitting the ciphertext

$$\tilde{C} := (C_{\mathsf{pub}}, C^{(i)} \oplus m').$$

2. If the decryption oracle $\mathcal{O}_{\mathsf{Dec}}$ responds with

$$\mathcal{O}_{\mathsf{Dec}}(\tilde{C}) = iv||0^{31}||1 + i, \quad (3)$$

   then the adversary concludes that $m' = m^{(i)}$.

To see that this indeed allows the attacker to determine whether $C^{(i)}$ encrypts m', note that in GCM-mode

$$\mathsf{Dec}(k, C^{(i)} \oplus m^{(i)}) = iv||0^{31}||1 + i$$

holds if and only if

$$C^{(i)} = \mathsf{Enc}(k, iv||0^{31}||1 + i) \oplus m^{(i)}.$$

Because $(\mathsf{Enc}, \mathsf{Dec})$ is a block-cipher, $\mathsf{Enc}(k, \cdot)$ is a permutation, and $\mathsf{Dec}(k, \cdot) = \mathsf{Enc}^{-1}(k, \cdot)$ is its inverse. Thus, if Equation (3) holds, then it must hold that $m^{(i)} = m'$.

2.2.3 Why Distinguishing Attacks Matter

Practitioners are prone to dismissing distinguishing attacks as being only of theoretical interest. However, we caution against this viewpoint, for two reasons. Firstly, such attacks are readily converted into plaintext recovery attacks when the plaintext is known to be of low entropy. We will demonstrate this in practice in Section 5.2. Secondly, such attacks are indicative of problems that tend to become more severe with time. The recent example of TLS1.0 provides a good example of this phenomenon: as early as 1995, Rogaway [62] pointed out that CBC encryption is vulnerable to a chosen plaintext distinguishing attack when the IVs used are predictable to the adversary. This vulnerability was addressed in TLS1.1, but TLS1.0 support remained widespread. Then in 2011, the Duong and Rizzo BEAST attack [25] showed how to extend Rogaway's original observation to produce a full plaintext recovery attack. Their attack applies to certain applications of TLS in which there is some adversarially-controllable flexibility in the position of unknown plaintext bytes. The resulting scramble to update implementations to avoid the Rogaway/BEAST attack could easily have been avoided had the distinguishing attack been given more credence in the first place.

2.2.4 Optimizations

We have based our description of the GCM distinguishing attack in Section 2.2.2 on the availability of an abstract CBC decryption oracle $\mathcal{O}_{\mathsf{CBC}}$. This oracle can be provided *somehow*, that is, by an arbitrary attack on CBC-mode encryption. The distinguishing attack uses the $\mathcal{O}_{\mathsf{CBC}}$ oracle naively

as a black-box, without taking into account which specific weaknesses of CBC-encryption and the target application are exploited to implement $\mathcal{O}_{\mathsf{CBC}}$. While on the positive side this implies that the GCM distinguishing attack works in combination with *any* CBC decryption attack, we also note that an attack making naive usage of the $\mathcal{O}_{\mathsf{CBC}}$ oracle is potentially not optimally efficient.

For instance, in practice the CBC decryption oracle is usually given by a padding oracle attack. A typical padding oracle attack requires on average between 14 [38] and 128 [68, 24] chosen-ciphertext queries to recover one plaintext byte. If the CBC decryption oracle $\mathcal{O}_{\mathsf{CBC}}$ is used naively as a black-box, without further consideration of which particular attack is performed by $\mathcal{O}_{\mathsf{CBC}}$, then this complexity is inherited by the attack on GCM. Thus, in order to test whether a particular GCM-encrypted ciphertext block $C^{(i)}$ contains a particular message m' (in case of a 16-byte block cipher like AES [2]) one expects that between $14 \cdot 16 = 224$ and $128 \cdot 16 = 2048$ chosen-ciphertext queries are required to perform one test.

We note that the GCM distinguishing attack can be improved dramatically by exploiting specific properties of the provided CBC padding oracle and the application. Jumping a bit ahead, our implementation of the GCM distinguishing attack (as described in Section 5.2) uses an optimized version of the naive attack from Section 2.2.2. This optimized attack takes into account specific details of the target application, like formatting of valid plaintexts and padding, which allows for much more efficient attacks. For the optimized attacks on GCM in XML Encryption and JOSE detailed in Section 5.2, only 2 queries are already sufficient to mount our distinguishing attack.

3 Further BC Attacks on Symmetric Cryptography and Generic Countermeasures

The principle of backwards-compatibility attacks on symmetric encryption schemes is of course not limited to CBC and GCM. We have chosen to describe this special case in the previous section as a first example, and because it represents a reasonable practical scenario which nicely matches the practical attacks described in Section 5.2. In this section, we discuss further BC attacks on symmetric encryption schemes and generic countermeasures.

3.1 BC Attacks on Other Modes of Operation

There exists a large number of block-cipher modes of operation defined by various organizations in various standards. For instance, popular unauthenticated modes of operation are ECB, CBC, OFB, and CTR [53, 55]. Widely used authenticated modes of operation are OCB [63], EAX [11], and CCM [56].

For any authenticated mode of operation, one can select a suitable unauthenticated mode of operation and describe a backwards compatibility attack which allows an attacker to distinguish encrypted messages or even to decrypt high-entropy ciphertexts. Since of course most combinations of modes of operation and attack scenarios are not of practical relevance, and the additional theoretical contribution over the attack from Section 2.2.2 is limited because the attack principle is always the same, we do not describe all possible attacks in detail.

We note only that different modes of operation have very different properties and characteristics w.r.t. backwards compatibility attacks. For example:

1. Some modes use the encryption algorithm $\mathsf{Enc}(k, \cdot)$ of the block-cipher for encryption, and the decryption algorithm $\mathsf{Dec}(k, \cdot)$ for decryption. Examples for such modes are ECB and CBC.
2. Some modes use the encryption algorithm $\mathsf{Enc}(k, \cdot)$ of the block-cipher for *both* encryption and decryption. Examples of this type are OFB and "counter"-modes, like CTR and GCM, where the block-cipher is turned into a stream cipher by encrypting an incrementing counter value.

The type of oracle provided by an attack on a mode of operation depends strongly on such characteristics. For instance, a CBC decryption attack provides a block-cipher *decryption* oracle that allows an attacker to compute the block-cipher decryption function $\mathsf{Dec}(k, \cdot)$. In contrast, a decryption attack on OFB mode would provide a block-cipher *encryption* oracle $\mathsf{Enc}(k, \cdot)$.

In Section 2.2.2 we have shown that the block-cipher decryption oracle $\mathsf{Dec}(k, \cdot)$ provided by the attack on CBC is sufficient to mount a distinguishing attack on GCM. In turn, this allows the decryption of low-entropy ciphertexts by exhaustive search over all possible plaintexts. If instead an *encryption* oracle was given, then this would even allow the decryption of high-entropy GCM ciphertexts, since this oracle essentially computes the block-cipher operation performed in the GCM-decryption algorithm.

In a different application scenario, with a different combination of algorithms, a block-cipher decryption oracle may also lead to a full-fledged decryption attack. For example, AES Key Wrap [54] is a NIST-specified symmetric key transport mechanism designed to encapsulate cryptographic keys. AES Key Wrap is used, for instance, in XML Encryption. Indeed, the block-cipher decryption oracle provided by known attacks [38] on XML Encryption allows to *decrypt* even high-entropy keys encrypted with the AES Key Wrap scheme.

3.2 Generic Countermeasures

There are a number of obvious countermeasures which would prevent our symmetric BC attacks. The cleanest approach is to fully embrace the principle of *key separation*, which dictates that different keys should be used for different purposes. Extending this principle would mean using completely different keys for different algorithms serving the same purpose. Of course, the required keys may not be readily available, and making them available might require significant re-engineering of other system components. This approach does not sit well with maintaining backwards compatibility.

A compromise position would be to take the existing key and ensure that distinct, algorithm-specific keys are derived from it using suitable algorithm identifiers. For example, we could set $k' = \mathsf{PRF}(k, \text{"Algorithm Identifier"})$ where now the original key k is used as a key to a pseudo-random function supporting key derivation. Suitable pseudorandom functions can be implemented based on block-ciphers or hash functions, which are readily available in most cryptographic libraries.

4 BC Attacks on Public-Key Cryptography

In this section, we recall the well-known attack of Bleichenbacher [13] on RSA-PKCS#1 v1.5 encryption [43]. We discuss its applicability to RSA-OAEP encryption [10] (as standardized in RSA-PKCS#1 v2.0 [44] and v2.1 [42]) and to RSA-PKCS#1 v1.5 signatures [42].

Essentially, Bleichenbacher's attack allows to invert the RSA-function $m \mapsto m^e \bmod N$ without knowing the factorization of N. This fact gives rise to obvious attacks on RSA-based encryption and signature schemes. Therefore the fact that Bleichenbacher's attack may in certain applications give rise to backwards compatibility attacks is not very surprising. We stress that we consider the contribution of this part of the paper therefore not in demonstrating this relatively obvious fact, but rather in showing that such attacks are indeed applicable in practice.

4.1 PKCS#1 v1.5 Padding and Encryption

In the sequel let (N, e) be an RSA public key, with corresponding secret key d. We denote with ℓ the byte-length of N, thus, we have $2^{8(\ell-1)} < N < 2^{8\ell}$.

The basic idea of PKCS#1 v1.5 [43] is to take a message k (a bit string), concatenate this message with a random padding string PS, and then apply the RSA encryption function $m \mapsto m^e \bmod N$. More precisely, a message k of byte-length $|k| \leq \ell - 11$ is encrypted as follows.

1. Choose a random padding string PS of byte-length $\ell - 3 - |k|$, such that PS contains no zero byte. Note that the byte-length of PS, denoted $|PS|$, is at least 8.
2. Set $m := \texttt{0x00}||\texttt{0x02}||PS||\texttt{0x00}||k$. Interpret m as an integer such that $0 < m < N$.
3. Compute the ciphertext as $c = m^e \bmod N$.

The decryption algorithm computes $m' = c^d \bmod N$ and interprets integer m' as a bit string. It tests whether m' has the correct format, i.e. whether m' can be parsed as $m' = \texttt{0x00}||\texttt{0x02}||PS||\texttt{0x00}||k$ where PS consists of at least 8 non-zero bytes. If this holds, then it returns k, otherwise it rejects the ciphertext.

4.2 Bleichenbacher's Attack

The only necessary prerequisite to execute Bleichenbacher's attack [13] is that an oracle $\mathcal{O}_{\mathsf{BB}}$ is given which tells whether a given ciphertext is valid (that is, PKCS#1 v1.5 conformant) with respect to the target public key (N, e). This oracle takes as input a ciphertext c and responds as follows.

$$\mathcal{O}_{\mathsf{BB}}(c) = \begin{cases} 1 \text{ if } c \text{ is valid w.r.t. PKCS\#1 v1.5 and } (N, e), \\ 0 \text{ otherwise.} \end{cases}$$

Such an oracle may be given in many practical scenarios, for instance by a web server responding with appropriate error messages. The applicability of Bleichenbacher's attack to XML Encryption – not only due to implementational issues, but also due to inherent properties of XML Encryption itself – was noticed in [37]. However, we stress that [37] considered only attacks on the legacy version v1.5 of PKCS#1 encryption. In this paper, we show that this weakness can also be used to break the security of current versions v2.0 and v2.1 of PKCS#1 (aka. RSA-OAEP) and to forge XML Signatures.

We give only a high-level description of the attack, and refer to the original paper [13] for details. Suppose a PKCS#1 v1.5 conformant ciphertext $c = m^e \bmod N$ is given. Thus, $m = c^d \bmod N$ lies in the interval $[2B, 3B)$, where $B = 2^{8(\ell-2)}$. Bleichenbacher's algorithm proceeds as follows. It chooses a small integer s, computes

$$c' = (c \cdot s^e) \bmod N = (ms)^e \bmod N,$$

and queries the oracle with c'. If $\mathcal{O}_{\mathsf{BB}}(c') = 1$, then the algorithm learns that $2B \leq ms - rN < 3B$ for some small integer r which is equivalent to

$$\frac{2B + rN}{s} \leq m < \frac{3B + rN}{s}.$$

By iteratively choosing new s, the adversary reduces the number of possible values of m, until only one is left.

For a 1024-bit modulus and a random ciphertext, the original analysis in [13] shows that the attack requires about one million oracle queries to recover a plaintext. Therefore, Bleichenbachers attack became also known as the "Million Message Attack". Recent improvements in cryptanalysis [7] show, however, that this number can be significantly improved. In particular, in certain (realistic) scenarios the improved attack of [7] performs only about 3800 oracle queries, depending on which ciphertext validity checks are performed by the oracle.

4.3 The Power of Bleichenbacher's Attack

As already noted in [13], the attack of Bleichenbacher allows not only to decrypt PKCS#1 v1.5 ciphertexts. Instead, it uses the PKCS#1 validity oracle to invert the RSA function $m \mapsto m^e \bmod N$ on an *arbitrary* value (not necessarily a PKCS#1 v1.5-conformant ciphertext).

Therefore Bleichenbacher's attack can potentially also be used to decrypt RSA-OAEP ciphertexts, or to forge RSA-based signatures, whenever the following two requirements are met.

1. The PKCS#1 v1.5 encryption scheme and the attacked cryptosystem (like RSA-OAEP encryption or RSA-signatures) use the same RSA-key (N, e).
2. A PKCS#1 v1.5-validity oracle is given, in order to mount Bleichenbacher's attack.

We will show that these two requirements are indeed met in certain practical applications, where PKCS#1 v1.5 encryption is available due to backwards compatibility reasons.

4.3.1 Attacking RSA-OAEP

The basic idea of RSA-OAEP (aka. PKCS#1 v2.0 [44] or 2.1 [42]) is very similar to PKCS#1 v1.5, except that a much more complex padding scheme is used.

Let us describe the padding in more detail. In the sequel let $\ell_G, \ell_H, \ell_k, \ell_0 \in \mathbb{N}$ be integers such that $\ell = 2 + \ell_G + \ell_H$ and $\ell_0 = \ell_G - \ell_k$. Moreover, let $G : \{0,1\}^{\ell_H} \to \{0,1\}^{\ell_G}$ and $H : \{0,1\}^{\ell_G} \to \{0,1\}^{\ell_H}$ be cryptographic hash functions.

A message k of bit-length ℓ_k is encrypted as follows.

1. Choose a random padding string $r \in \{0,1\}^{\ell_H}$.
2. Compute values $s \in \{0,1\}^{\ell_G}$ and $t \in \{0,1\}^{\ell_H}$ as

$$s := k||0^{\ell_0} \oplus G(r) \quad \text{and} \quad t := r \oplus H(s).$$

3. Set $m := 02||s||t$. Interpret m as an integer such that $0 < m < N$.
4. Compute the ciphertext as $c = m^e \bmod N$.

Note that in order to decrypt an OAEP-ciphertext it suffices to be able to invert the RSA encryption function $m \mapsto m^e \bmod N$, since the message encoding and decoding steps are unkeyed. Thus, if the RSA public key (N, e) is used for OAEP-encryption and an oracle $\mathcal{O}_{\mathsf{BB}}$ is available which tells whether a given ciphertext is PKCS#1 v1.5 conformant w.r.t. (N, e), then one can use this oracle to decrypt OAEP-ciphertexts by mounting Bleichenbacher's attack.

4.3.2 Attacking RSA-PKCS#1 v1.5 Signatures

In the sequel let $H : \{0,1\}^* \to \{0,1\}^{8\ell_H}$ be a cryptographic hash function (e.g. SHA-1) with ℓ_H-byte output length. Let (N, e) be an RSA public key, where N has byte-length ℓ, with corresponding secret key $d = 1/e \bmod \phi(N)$. A digital signature over message m according to RSA-PKCS#1 v1.5 is computed in three steps.

1. Compute the hash value $H(m)$.
2. Prepend $H(m)$ (from right to left) with
   - a 15-byte string $\mathsf{ASN.1}$, which identifies the hash function H,
   - one `0x00`-byte,
   - $\ell - \ell_H - 17$ copies of the `0xFF`-byte, and
   - the `0x01`-byte,

   to obtain a padded message string M of the form

$$M = \texttt{0x01}||\texttt{0xFF}||\ldots||\texttt{0xFF}||\texttt{0x00}||\mathsf{ASN.1}||H(m).$$

3. Compute the signature σ as

$$\sigma := M^d \bmod N.$$

Note that in order to forge an RSA-PKCS#1 v1.5 signature it suffices to be able to invert the RSA encryption function. Thus, if the RSA public key (N, e) is used for RSA-PKCS#1 v1.5 signatures and an oracle $\mathcal{O}_{\mathsf{BB}}$ is available that tells whether a given ciphertext is PKCS#1 v1.5 conformant w.r.t. (N, e), then one can use this oracle to forge RSA-PKCS#1 v1.5 signatures by mounting Bleichenbacher's attack on a suitably randomized version of the encoded message M.

This attack possibility is mentioned in Bleichenbacher's original paper [13]. A variant of the attack was recently explored in [21] in the context of EMV signatures (where the same RSA key pair may be used for both signature and encryption functions).

4.4 Countermeasures and the Difficulty of Key Separation with X.509 Certificates

Key separation means to use different (independent) keys for different algorithms. In theory this principle provides a simple solution to prevent backwards compatibility attacks. As described in Section 3.2, key separation is very easy to enforce in the symmetric setting, for instance by a suitable application of a pseudorandom function before using the symmetric key.

In principle, key separation in the public-key setting is almost as easy to enforce as in the symmetric setting. One

could simply generate different keys for different purposes. For instance, one RSA-key (N_0, e_0) is generated exclusively for PKCS#1 v1.5 encryption, another independent RSA-key (N_1, e_1) exclusively for PKCS#1 v1.5 signature, and yet another independent RSA-key (N_2, e_2) only for RSA-OAEP encryption. Each public-key should then be published together with some information (included in the certificate, for instance) that specifies for which algorithm this key can be used. Accordingly, each secret key should be stored together with this additional information. Cryptographic implementations should check whether the provided key is suitable for the executed algorithm.

Unfortunately this theoretically sound solution is not easy to implement in practice. This is because common data formats for public keys do not provide this additional information as part of the basic standard. For example, the X.509 standard for public-key certificates defines a popular data format for public keys. While an X.509 certificate does include algorithm identifiers for the signing algorithm used to create the certificate itself, these certificates do not necessarily include any information about with which algorithms the certified public key can be used. It is possible to extend X.509 certificates with such a field, the Subject Public Key Info field (see RFC 5280 [19] and more specifically RFC 4055 [65] for naming conventions for RSA-based algorithms), but supporting this field is not mandatory and would require major changes to implementations and libraries. In view of BC attacks, we consider this to be a big handicap of X.509 certificates. We suggest that algorithm identifiers for certified keys be included by default in future revisions of X.509.

The importance of key separation still seems to be not very well understood in practice. For instance, a large cloud identity security provider even *suggests* the use of RSA keys for both digital signatures and encryption [60], while RFC 4055 [65] permits the same RSA key pair to be used for more than one purpose (see specifically Section 1.2 of RFC 4055). There is limited theoretical support for this kind of key reuse (see [58] and the references therein), but in general, as our attacks show, it is a dangerous practice.

5 Practical BC attacks on XML Encryption and JSON Web Encryption

In this section we demonstrate the vulnerability of current versions of XML Encryption [28] and JSON Web Encryption [41] to BC attacks. We first give a brief overview of these standards. Then we describe optimized versions of the BC attacks illustrated in previous sections. Finally we discuss practical countermeasures, their applicability, and propose changes to the algorithms and security considerations in the analyzed standards.

5.1 Technical Background

XML Encryption [28] and JSON Web Encryption [41] are two standards that specify a way to apply cryptographic algorithms to data transported over a network. Both standards provide security at the message-level. In this section we describe their main properties, as far as they are relevant to our attacks.

5.1.1 XML Encryption and XML Signature

The Extensible Markup Language (XML) [14] defines a structure for flexible storage and transmission of tree-based data. It is widely used for instance in Single Sign-On [15] scenarios and in Web Services [50]. The wide adoption of XML has raised the demand for security standards enabling the application of encryption and digital signatures to XML documents. This lead to the introduction of XML Encryption [28] and XML Signature [29].

The increasing adoption of XML Signature and XML Encryption in enterprise applications is confirmed by a large number of commercially available XML gateways [35, 49, 57] and enterprise software [39, 64] supporting these standards. Both are implemented (or being implemented) in a wide range of systems processing sensitive data, including banking [20], eGovernment [34, 46, 69], and eHealth infrastructures [16, 18].

XML Encryption. In order to encrypt XML data in common Web Services scenarios, usually *hybrid encryption* is used. That is, encryption proceeds in two steps:

1. The sender chooses a *session key* k. This key is encrypted with a public-key encryption scheme, under the receiver's public-key, which yields a ciphertext C_{pub}.
2. The actual payload data is then encrypted with a symmetric encryption algorithm using the key k, yielding a ciphertext C_{sym}.

The XML Encryption W3C standard [27] specifies two public-key encryption schemes, namely PKCS#1 in versions 1.5 and 2.0. Both are mandatory to implement. Furthermore, the standard allows to choose between three symmetric ciphers: AES-CBC, 3DES-CBC, or AES-GCM. AES-CBC and 3DES-CBC have been a part of this standard since its earliest version. AES-GCM was included in the latest standard draft version in order to prevent a recent attack on CBC-based XML Encryption [38]. AES-CBC and 3DES-CBC are still included in the standard, for backwards compatibility reasons. All the three algorithms are mandatory to implement for each standard-conformant service.

In the context of Web Services, XML Encryption ciphertexts are transported in SOAP [33] messages. Figure 3 gives

```
<Envelope>
 <Header>
  <Security>
   <EncryptedKey Id="EncKeyId">
    <EncryptionMethod Algorithm="..xmlenc11#rsa-oaep"/>
    <KeyInfo>
     <SecurityTokenReference>
      <KeyIdentifier>...<KeyIdentifier>
     </SecurityTokenReference>
    </KeyInfo>
    <CipherData>
     <CipherValue>Y2bh...fPw==</CipherValue>
    </CipherData>                                   c_pub
   </EncryptedKey>
  </Security>
 </Header>
 <Body>
  <EncryptedData Id="EncDataId-2">
   <EncryptionMethod Algorithm="..xmlenc11#aes128-gcm"/>
   <CipherData>
    <CipherValue>3bP...Zx0=</CipherValue>
   </CipherData>                                    c_sym
  </EncryptedData>
 </Body>
</Envelope>
```

Figure 3. Example of a SOAP message with encrypted data consisting of two parts: C_{pub} encrypting the symmetric key k using RSA-OAEP and C_{sym} encrypting the actual payload using AES-GCM.

an example of a SOAP message containing a message encrypted according to XML Encryption. In this example the sender uses PKCS#1 v2.0 in combination with AES-GCM.

XML Signature. Along with XML Encryption, the W3C Working Group defined the XML Signature standard [29], which allows to protect integrity and authenticity of XML messages.

In order to describe our attacks, it is not necessary to describe the XML Signature standard in detail. It is sufficient to know that in most application scenarios the RSA-PKCS#1 v1.5 signature scheme is used.

Platform for Experimental Analysis. We analyze the practicality and performance of our attacks on XML Encryption and XML Signature by applying them to the Apache Web Services Security for Java (Apache WSS4J) library. This is a widely used library providing Web Services frameworks with implementations of XML Encryption and XML Signature. It is used in several major Web Services frameworks, including JBossWS [39], Apache CXF [5], and Apache Axis2 [4].

5.1.2 JSON Web Encryption and Signature

JavaScript Object Notation (JSON) is a lightweight text-based standard for description and exchange of arbitrary data. The JSON Web Encryption (JWE) [41] and JSON Web Signature (JWS) [40] standards are maintained by the Javascript Object Signing and Encryption (jose) Working Group. These standards are quite recent, with the first public draft dating to January 2012.

```
{"alg":"RSA1_5",
 "enc":"A256GCM",
  "iv":"__79_Pv6-fg",
 "jku":"https://example.com/p_key.jwk"}
```

Figure 4. JSON Web Encryption header segment example specifying encryption algorithms.

JSON Web Encryption. JSON Web Encryption (JWE) specifies how to apply encryption schemes to JSON data structures. JWE supports different methods for data encryption, using symmetric and public-key encryption algorithms. The current draft 06 of the JWE standard includes the algorithms AES-CBC with HMAC, AES-GCM, and AES-KeyWrap as mandatory symmetric ciphers. The mandatory public-key encryption schemes are PKCS#1 v1.5 and v2.0 encryption.

A JSON Web Encryption message consists of two components. The *body segment* contains a ciphertext encrypting the payload data. The *header segment* contains information about the algorithms used to encrypt this ciphertext contained in the body. An example of a JWE header segment is given in Figure 4. In this example RSA-PKCS#1 v1.5 is used to encapsulate a symmetric key. The actual payload data is encrypted under this key using AES-GCM.

JSON Web Signature. Different methods to secure integrity and authenticity of JSON messages are provided by the JSON Web Signature (JWS) [40] standard. Again, in order to describe our attacks it is sufficient to know that the JSON Web Signature standard includes the RSA-PKCS#1 v1.5 signature scheme.

Platform for Experimental Analysis. We investigate the practicality and performance of our attacks on JWE and JWS by applying them to the Nimbus-JWT library [52]. Nimbus-JWT is a Java implementation of JSON Web Encryption (JWE) and JSON Web Signature (JWS), developed by NimbusDS to support their Cloud Identity management portfolio.

Even though Nimbus-JWT claims to implement version 02 of the JWE standard draft, it still supports usage of AES-CBC (without MAC), which was available in version 01, but not in version 02 or any subsequent versions.

5.1.3 Analysis on the Library Level

Note that we test our attacks at the library level, not against actual applications. It may therefore be possible that applications implement specific countermeasures to prevent these attacks. However, we stress that preventing most attacks on higher application layers is extremely difficult or even impossible, as we describe later in this section.

5.2 Breaking AES-GCM

In this section, we describe our practical attacks breaking indistinguishability of the AES-GCM ciphertexts in XML Encryption. We first describe a performant variant of the attack from Section 2. Then we present the results of our experimental evaluation, executed against Apache WSS4J and, for completeness, against the Nimbus-JWT library.

5.2.1 Plaintext Validity Checking

When processing a symmetric XML Encryption ciphertext, an XML library typically proceeds as follows. It takes a symmetric decryption key and decrypts the ciphertext. Then the validity of the padding is checked and the padding is removed. Finally, the decrypted plaintext is parsed as XML data. If any of these steps fails, the process returns a processing failure.

In this section we describe how padding scheme and parsing mechanisms in XML Encryption work. They build an important ingredient to our optimized attack.

In the sequel let us assume that XML Encryption is used with a block-cipher of block size $bs = 16$ byte, like AES, for instance.

Padding in XML Encryption. XML Encryption uses the following padding scheme:

1. The smallest non-zero number $plen$ of bytes that must be padded to the plaintext m to achieve a multiple of the block size is computed.
2. $plen - 1$ random padding bytes are appended to m.
3. $plen$ is interpreted as an integer and appended to m.

For instance, when using a block cipher with 16-byte block size, a 10-byte plaintext block m would be padded to $m' = m||pad$, where:

$$pad = \texttt{0x??????????06}.$$

Observe that a randomly generated plaintext block is valid according to the XML Encryption padding scheme with a probability of $P_{pad} = 16/256$ (if a 16-byte block cipher is used, as we assume), since there are 16 possible values for the last byte that yield a valid padding.

XML Parsing. XML is a structured representation of data. Valid XML plaintexts have to consist of valid characters and have a valid XML structure. The XML Encryption standard prescribes that characters and symbols are encoded according to the UTF-8 [72] code. Parsing XML data that are not well-formed or contain invalid characters will lead to parsing errors.

Note that the first 128 characters in UTF-8 are identical to the American Standard Code for Information Interchange (ASCII) [17]. For simplicity, let us assume in the following that an XML plaintext consists only of ASCII characters. The ASCII code represents characters as single bytes, and allows to encode $2^7 = 128$ different characters.

As the ASCII table includes only 128 characters, the first bit of a byte representing an ASCII character is always equal to 0. Another characteristic of the ASCII table is that it contains two sets of characters: parsable and non-parsable [38]. Parsable characters include letters, numbers, or punctuation marks. About a 25% of ASCII characters are non-parsable. This includes, for example, the `NUL`, `ESC`, and `BEL` characters. If any of these is contained in an XML document, then this will lead to a parsing error.

Thus, P_{parse}, the probability that a random byte corresponds to a parsable character, is about $1/2 \cdot 3/4 = 3/8$.

Probability of valid XML ciphertexts. The fact that an XML processor responds with an error message if the padding or the plaintext format of a decrypted message is invalid allows us to determine whether a given CBC-encrypted ciphertext is valid or not. This allows us to construct an XML decryption validity oracle $\mathcal{O}_{\mathsf{CBCxml}}$, which takes as input an AES-CBC ciphertext $\tilde{c} = (\tilde{iv}, \tilde{C}^{(1)})$, decrypts it, and responds with 1 if the plaintext is correct, and 0 otherwise.

In particular, a randomly generated ciphertext $(\tilde{iv}, \tilde{C}^{(1)})$ consisting of an initialization vector and one ciphertext block leads to a decryption error with high probability. The probability that a random ciphertext is valid is only

$$P_{valid} = \sum_{i=0}^{15} (1/256)(3/8)^i \approx 0.0062$$

This low probability that a random ciphertext is valid is one of the key ingredients to our attack.

Plaintext Validity Checking in JWE. The JWE standard applies a different padding scheme, namely PKCS#5. This padding scheme has a more restrictive padding validity check, such that random ciphertexts are rejected with even higher probability. This improves the success probability of our attack. In the context of JWE we thus obtain a plaintext validity oracle $\mathcal{O}_{\mathsf{CBCjwe}}$, which is similar to $\mathcal{O}_{\mathsf{CBCxml}}$ but has an even smaller false positive rate.

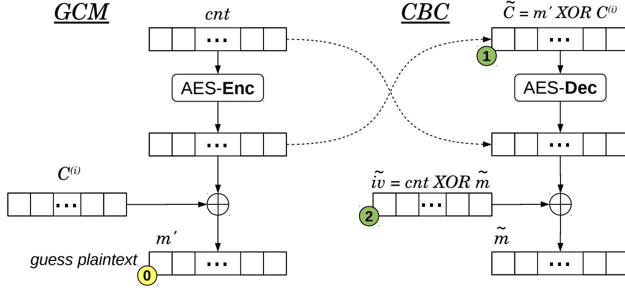

Figure 5. Breaking indistinguishability of AES-GCM with AES-CBC

5.2.2 Optimized Algorithm

Distinguishing Plaintexts. Let us now describe our optimized attack. Consider an attacker who eavesdrops an AES-GCM ciphertext

$$C = (iv, C^{(1)}, \dots, C^{(n)}, \tau).$$

His goal is to determine whether the i-th ciphertext block $C^{(i)}$ encrypts a particular message m'. The attacker proceeds as follows (see Figure 5):

1. He computes a CBC ciphertext by setting the first ciphertext block equal to $\tilde{C} = m' \oplus C^{(i)}$. (If he has guessed m' correctly, then this sets $\mathsf{Dec}(k, \tilde{C}) = cnt = iv||0^{31}||1+i$.)
2. He selects a valid XML plaintext $\tilde{m}$ and a CBC-mode initialization vector $\tilde{iv}$, such that

$$\tilde{m} = \tilde{iv} \oplus cnt$$

3. The ciphertext $(\tilde{iv}, \tilde{C})$ is then sent to the CBC validity checking oracle.

If the CBC validity checking oracle accepts this as a valid ciphertext, then the attacker has most likely guessed m' correctly (with a probability of $P_{m'} = 1 - P_{valid} \approx 0.9938$). Otherwise, he has certainly guessed wrongly. This test can be repeated a few times with distinct values of $\tilde{m}$ to decrease the probability of a false positive.

Recovering Plaintext Bytes. The distinguishing attack can also be used to decrypt low-entropy plaintexts. For our experiments, we consider an attacker that *a priori* knows the complete plaintext except for one plaintext byte. We also assume that the attacker reduces the number of false positives by one additional oracle query for each positive response.

The attack procedure for recovering one plaintext byte is depicted in Algorithm 1. The algorithm iterates over all the $n = 256$ possibilities for byte b. The performance of this step can be improved significantly if the attacker is able to narrow the number of possible values for b, for instance if the target application accepts only ASCII characters, only letters, only integers, etc.

Algorithm 1 Recovering a single plaintext byte b from an AES-GCM ciphertext using an $\mathcal{O}_{\mathsf{CBCxml}}$ oracle.

Input: Plaintext block m' containing one unknown byte b. Position p of the unknown byte b. AES-GCM ith ciphertext block $C^{(i)}$ and initialization vector iv.
Output: Plaintext byte b.

1: $\tilde{m}_{valid1}$:= `0x00`$||$`0x00`$|| \dots ||$`0x00`$||$`0x10`
2: $\tilde{m}_{valid2}$:= `0x40`$||$`0x40`$|| \dots ||$`0x40`$||$`0x01`
3: $cnt := iv||0^{31}||1+i$
4: $n := 256$
5: **for** $b = 0 \to (n-1)$ **do**
6: $m'[p] := b$
7: $\tilde{C} := m' \oplus C^{(i)}$
8: $\tilde{iv} := cnt \oplus m_{valid1}$
9: **if** $\mathcal{O}_{\mathsf{CBCxml}}(\tilde{iv}, \tilde{C}) = 1$ **then**
10: $\tilde{iv} := cnt \oplus m_{valid2}$
11: **if** $\mathcal{O}_{\mathsf{CBCxml}}(\tilde{iv}, \tilde{C}) = 1$ **then**
12: **return** b
13: **end if**
14: **end if**
15: **end for**

The algorithm can easily be extended to decrypt larger numbers of unknown bytes in one block. To decrypt x unknown bytes, the attacker needs to issue about n^x oracle queries.

5.2.3 Evaluation

We evaluated performance of our attacks against both WSS4J and Nimbus-JWT. We first used the libraries to generate valid messages containing AES-GCM ciphertexts. Then we modified the algorithm parameters in the messages, forcing the receiver to process the ciphertexts using AES-CBC, and executed the attack described in Algorithm 1. The required ciphertext validity oracles were based on error messages generated by the libraries.

Table 1 reports the results of our evaluation, with figures obtained by averaging over 50 executions. We include results for ciphertext blocks containing 1, 2, and 3 unknown bytes. We restricted the possible character set to a group of alphabetic and numeric characters. Thus, in this setting the attacker needs to test $n = 64$ possibilities for each byte.

As expected, the attack performs well if the target ciphertext blocks contain a large number of known plaintext bytes. The number of oracle queries needed increases exponentially with the number of unknown plaintext bytes.

While the number of guessed m' plaintexts is constant for both libraries, the number of total oracle queries varies.

Number of unknown bytes	Guessed m' plaintexts	$\mathcal{O}_{\text{CBCxml}}$ queries	$\mathcal{O}_{\text{CBCjwe}}$ queries
1	36	37	37
2	2,130	2,145	2,139
3	142,855	143,811	143,409

Table 1. Attack results applied on ciphertext blocks containing 1, 2, and 3 unknown bytes from a group of alphabetic and numeric characters.

The different numbers of queries is caused by different plaintext validation models being used in the XML Encryption and JWE standards: the validation model in JWE employes a stricter verification for the padding, which results in less oracle queries being needed by the attacker.

Extension to a Full Plaintext Recovery Attack. Our evaluation shows that an attacker is able to efficiently decrypt ciphertexts with a large number of known bytes in the plaintext. We note that an attacker who is able to control parts of the plaintext is also able to recover high-entropy plaintexts, by employing the technique from Duong and Rizzo's BEAST attack [25].

Let us sketch the basic idea of this technique. The attacker uses its control over the plaintext to prepend the unknown high-entropy plaintext with $n-1$ known bytes, where n is the block-size of the block cipher in bytes. Thus, only the last byte of the first block is unknown to the attacker, and can be recovered relatively efficiently with the above procedure. In the next step, the attacker prepends the high-entropy plaintext with $n-2$ known bytes. Since the first byte of the plaintext is already recovered, there is again only one unknown byte in the resulting plaintext. By executing Algorithm 1 repeatedly with this divide-and-conquer strategy, the attacker is able to recover the full high-entropy plaintext.

5.3 Practical Examples of BC Attacks on Public-Key Cryptography

As described in Sections 5.1.1 and 5.1.2, both XML Encryption and JWE specify public-key encryption according to PKCS#1 v1.5 and v2.0 as being mandatory. Similarly, both XML Signature and JWS specify PKCS#1 v1.5 signatures as being mandatory.

Recall from Section 4.3 that the known attack of Bleichenbacher on PKCS#1 v1.5 can be used to decrypt PKCS#1 v2.0 ciphertexts or to forge RSA-signatures if two requirements are met:

1. The application allows the RSA public-key (N, e) used for PKCS#1 v2.0 encryption (or RSA-signatures) to be also used for PKCS#1 v1.5 encryption; and
2. the application provides a PKCS#1 v1.5 validity oracle.

It was recently observed [37] that both XML Encryption and JWE inherently provide PKCS#1 v1.5 validity oracles.[9] Thus, Property 2 is met by XML Encryption and JWE.

It remains to show that Property 1 is also met. Indeed, neither standard distinguishes between keys for PKCS#1 v2.0 encryption, PKCS#1 v1.5 encryption, and PKCS#1 v1.5 signatures (as noted before, some providers even recommend re-use of RSA-keys across different algorithms).

Let (N, e) be the RSA public key of a receiver. A ciphertext according to PKCS#1 (regardless of v1.5 or v2.0), consists of a single integer y modulo N. Thus, in order to apply the correct decryption algorithm to y, the receiver needs additional information, namely the version (v1.5 or v2.0) of PKCS#1 according to which the ciphertext c was encrypted by the sender. In both XML Encryption and JWE, this information is provided in metadata[10] accompanying the ciphertext. These metadata are (typically) not integrity-protected. Thus, an attacker can change them arbitrarily.

This enables an attacker to trick the receiver into applying the PKCS#1 v1.5 decryption algorithm to an arbitrary value y modulo N. In combination with the PKCS#1 v1.5 validity oracle from [37] and Bleichenbacher's attack [13], this suffices to invert the RSA-function $m \mapsto m^e \bmod N$ on an arbitrary value y. This in turn allows to decrypt PKCS#1 v2.0 ciphertexts or to forge RSA-signatures with respect to key (N, e), as explained in Section 4.3.

Experimental Results. In order to assess the practicability and performance of the attack, we implemented Bleichenbacher's attack on XML Encryption [13, 37] and applied it to the Nimbus-JWT library. The PKCS#1 v1.5 validity oracle was provided by exceptions thrown by this library.[11]

The experiment was repeated 10,000 times, each time with a fresh 1024-bit RSA-key, which was generated using the standard Java key pair generator.[12] Decrypting a random PKCS#1 v2.0 ciphertext took about 171,000 oracle queries on average. Forging a JSON Web Signature for an arbitrary message took about 218,000 queries on average. See

[9] Typically PKCS#1 v1.5 validity oracles are a result of careless implementations, provided by timing differences or distinguishable error messages. A noteworthy aspect of [37] is, that the availability of these validity oracles is not (only) an implementational issue, but an inherent property of both standards. This is a consequence of the way in which PKCS#1 v1.5-based public-key encryption is combined with CBC-based symmetric encryption, see [37] for details.

[10] The `EncryptedKey` element in XML Encryption, the *header segment* in JWE.

[11] In practice one would instead use the more elaborate attack techniques of [37] to determine whether a given ciphertext is PKCS#1 v1.5 valid.

[12] `java.security.KeyPairGenerator`.

	Mean	Median	Maximum # of queries	Minimum # of queries
PKCS#1 v2.0 Ciphertext	171,228	59,236	142,344,067	4,089
PKCS#1 v1.5 Signature	218,305	66,984	395,671,626	20,511

Table 2. Experimental results of BC attacks on PKCS#1 v2.0 ciphertexts and PKCS#1 v1.5 signatures.

Table 2 for details.

Executing the attacks with 2048 and 4096-bit RSA-keys resulted in only a slightly higher number of requests.

Improvements. Very recently, Bardou et al. [7] have described significantly improved variants of Bleichenbacher's attack that allow to reduce the number of oracle queries dramatically. We did not implement these optimizations yet, but since the improvements in [7] are very general, we expect that they lead to much more efficient BC attacks, too.

5.4 Practical Countermeasures

In this section we discuss why several seemingly simple countermeasures (cf. Sections 3.2 and 4.4) are hard to employ in practice.

5.4.1 Unifying Error Messages

In our experimental analysis, we applied BC attacks on the library level by exploiting (relatively detailed) error messages returned by the decryption algorithm. One obvious approach to prevent attacks based on such detailed error messages is to suppress all error messages on the application level, hoping that an attacker that does not receive any information about the reason for a decryption failure (incorrect decryption, invalid plaintext format, etc.) will not be able to mount the attack with reasonable efficiency.

However, we note that there exist several other additional side-channels turning servers into validity oracles that enable efficient attacks, even if the server responds with a unified error message. First, it has been shown that by attacking XML Encryption in Web Services an attacker can determine if a ciphertext contains a valid plaintext or not even if the Web Service returns only two types of responses (valid or invalid) by employing a technique called XML Encryption wrapping [67]. This technique can be applied to symmetric as well as asymmetric ciphertexts, and has proven to be practical when applied to major Web Services frameworks, even if the messages are authenticated with XML Signatures. We believe similar attacks can be executed against JWE libraries, too.

Second, there may be further side channels. A classical example is different timing behaviour in case of different errors, which again would allow to distinguish which type of error has occurred [37].

5.4.2 Disallowing Legacy Algorithms

Another obvious countermeasure would be to disallow all legacy algorithms and to use only state-of-the-art cryptosystems. Unfortunately, this countermeasure would also destroy interoperability for all parties that are only capable of running older algorithms. This is a real issue: for example, the attack on XML Encryption from [38] showed the insecurity of CBC-mode in XML Encryption. Therefore GCM is now available as an additional option in the standard. Even though the attack was published almost one year ago (and was disclosed to vendors and developers several months earlier), users applying important Web Services frameworks like Apache Axis2 [4] or SAML-based Single Sign-On [15] frameworks like Shibboleth [66] cannot adapt GCM as the platforms these frameworks are running on do not support GCM.

In the case of XML Encryption and Web Services one may also apply WS-Security Policy [48]. This standard allows the definition of policies forcing usage of specific algorithms in client-server communication. However, it is still questionable how strictly these policy restrictions are implemented. We present some details about the implementation of this standard in Apache CXF in Section 5.5.

5.4.3 Key Separation

Symmetric Algorithms. The *key separation* countermeasures proposed in Section 3.2 is simple and effective. As the JWE standard is still in a draft version, we strongly recommend to consider application of this principle in the final version of JWE. To our knowledge, the implementation of key separation is currently under discussion in the XML Encryption Working Group, motivated by the attacks presented in this paper.

Asymmetric Algorithms. The key separation principle can also prevent BC attacks on public-key schemes like PKCS#1 v2.0. Unfortunately, it seems that the importance of this principle is not well-understood in practice. For instance, the WS-Security Policy standard [48] explicitly mentions in Section 7.5 that it is possible to use the same RSA key pair for encryption and signature processing. Moreover, some providers *suggest* their users to use the same RSA key pair for different cryptographic algorithms [31, 60].

We have learned that the XML Encryption Working Group will include considerations about key separation mechanisms in the XML Encryption standard.

5.5 Communication with Developers

We discussed our attacks with developers of several frameworks. In this section we summarize some approaches that developers have followed to counter them.

The most recent draft of XML Encryption which includes AES-GCM is not widely adopted yet (note that the first public version dates to March 2012). The only framework we are aware of that currently supports this version is Apache CXF [5].

5.5.1 Apache CXF and WSS4J

Apache CXF is one of the Web Services frameworks utilizing the tested Apache WSS4J library [6].

WS-Security Policy. One possibility to restrict the list of algorithms that can be used by Web Services is provided by the WS-Security Policy standard [48]. This standard allows the server to define specific algorithms that clients must use. Apache CXF supports the WS-Security Policy standard and correctly checks the algorithms used in the encrypted XML messages. However, the specific design of the Apache CXF framework means that algorithms used for data decryption are checked *after* the message is decrypted. This means the adversary is able to force the server to decrypt arbitrary data with arbitrary cryptographic algorithms, which in turn allows to use the server as an plaintext/ciphertext validity oracle, as required for our attacks.

The Apache CXF developers are now redesigning Apache WSS4J and Apache CXF implementations to check the used security algorithms *before* ciphertexts are decrypted.

Decrypting only signed elements. Another countermeasure thwarting our attacks is to process only those encrypted elements that were signed by XML Signatures [67]. Apache WSS4J library includes a configuration parameter called `REQUIRE_SIGNED_ENCRYPTED_DATA_ELEMENTS`. If this is set to `true`, then any symmetrically encrypted `EncryptedData` elements that are not signed are rejected without processing. In the default configuration, this parameter is set to `false`.

The developers have considered to default this parameter to `true` for `EncryptedData` elements secured with the CBC mode encryption in the next framework release. However, they have decided against this modification as it would break many existing use-cases.

5.5.2 Ping Identity

Ping Identity [36] is an identity management solution provider supporting SAML-based Single Sign-On [15]. It provides their customers with products such as PingFederate that can play the role of an Identity Provider (which authenticates identities and issues SAML tokens) or a Service Provider (which validates SAML tokens and signs in to integrated systems). Both products enable users to apply XML Encryption.

In its documentation, Ping Identity suggested its users could use the same asymmetric key pair for signature as well as encryption processing [60]. We notified the framework developers. The Ping Identity website was updated immediately and the suggestion removed. Moreover, we cooperated with the developers and evaluated XML Encryption processing in their Service provider and Identity provider implementations. We found that our BC attacks were applicable to the Service provider implementation in all the provided settings. The BC attacks against the Identity provider implementation could be executed for specific settings where XML Signatures are not applied.[13]

The Ping Identity developers have changed their implementation such that only *signed* XML ciphertexts will be decrypted. This will be available in the next release of their product. Furthermore, the RSA PKCS#1 v1.5 algorithm will be disabled by default *for message creators* (senders). For interoperability reasons receivers will still need to support RSA PKCS#1 v1.5. Even though the latter still enables BC attacks, this is a step towards phasing out RSA PKCS#1 v1.5.

5.5.3 Shibboleth

Shibboleth [66] is a SAML-based framework supporting federated identity management deployments. Decryption of XML messages is supported only in the Service provider implementation. XML Encryption is enabled by default in the Shibboleth deployments.

[13] The attacks against the Identity provider are significant, since they allow an attacker to forge Identity provider signatures for arbitrary SAML tokens when the same key pair for signature and encryption processing is used.

After we communicated the attacks to the framework developers, they decided to blacklist RSA PKCS#1 v1.5 by default in the newest Service provider version (Shibboleth 2.5.0).

6. Conclusions

We explored *backwards compatibility* attacks, which arise when a cryptographic standard offers a choice between several algorithms to perform the same cryptographic task and when some of those algorithms have known vulnerabilities. Our main point is that the mere presence of these insecure options can adversely affect the security of state-of-the-art algorithms, which would otherwise be invulnerable to attack. We demonstrated this point by describing practical attacks on the current versions of two important cryptographic standards, namely W3C's XML Encryption and JSON Web Encryption. We proposed practical and effective countermeasures that thwart these backwards compatibility attacks. Our attacks highlight a lack of appreciation for the principle of key separation in real world deployments of cryptography, and bring to the surface weaknesses in current standards for digital certificates.

Acknowledgements

We would like to thank Christopher Meyer for helpful discussions and the anonymous reviewers for providing helpful comments. We would also like to thank all mentioned vendors and the W3C Working Group for their cooperation and discussions on our attacks. Especially we would like to thank Scott Cantor, Colm O hEigeartaigh, and Yang Yu.

References

[1] T. Acar, M. Belenkiy, M. Bellare, and D. Cash. Cryptographic agility and its relation to circular encryption. In H. Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 403–422. Springer, May 2010.

[2] Advanced encryption standard (AES). National Institute of Standards and Technology (NIST), FIPS PUB 197, U.S. Department of Commerce, Nov. 2001.

[3] N. AlFardan and K. G. Paterson. Plaintext-recovery attacks against Datagram TLS. In *Network and Distributed System Security Symposium (NDSS 2012)*, 2012.

[4] Apache Software Foundation. Apache Axis2. `http://axis.apache.org/axis2/java/core`.

[5] Apache Software Foundation. Apache CXF. `http://cxf.apache.org`.

[6] Apache Software Foundation. Apache WSS4J - Web Services Security for Java, May 2012. `http://ws.apache.org/wss4j/`.

[7] R. Bardou, R. Focardi, Y. Kawamoto, G. Steel, and J.-K. Tsay. Efficient Padding Oracle Attacks on Cryptographic Hardware. In R. Canetti and R. Safavi-Naini, editors, *Advances in Cryptology – CRYPTO*, 2012.

[8] E. Barkan, E. Biham, and N. Keller. Instant ciphertext-only cryptanalysis of GSM encrypted communication. In D. Boneh, editor, *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 600–616. Springer, Aug. 2003.

[9] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In T. Okamoto, editor, *Advances in Cryptology – ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 531–545. Springer, Dec. 2000.

[10] M. Bellare and P. Rogaway. Optimal asymmetric encryption. In A. D. Santis, editor, *Advances in Cryptology – EUROCRYPT'94*, volume 950 of *Lecture Notes in Computer Science*, pages 92–111. Springer, May 1994.

[11] M. Bellare, P. Rogaway, and D. Wagner. The EAX mode of operation. In B. K. Roy and W. Meier, editors, *Fast Software Encryption – FSE 2004*, volume 3017 of *Lecture Notes in Computer Science*, pages 389–407. Springer, Feb. 2004.

[12] S. Bellovin. Problem areas for the IP security protocols. In *Proceedings of the Sixth Usenix Unix Security Symposium*, pages 1–16, July 1995.

[13] D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In H. Krawczyk, editor, *Advances in Cryptology – CRYPTO'98*, volume 1462 of *Lecture Notes in Computer Science*, pages 1–12. Springer, Aug. 1998.

[14] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition). *W3C Recommendation*, 2008.

[15] S. Cantor, J. Kemp, R. Philpott, and E. Maler. Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V2.0. OASIS Standard, March 2005.

[16] Centers for Disease Control and Prevention. Public Health Information Network (PHIN) – Secure Message Transport Guide, July 2008. Version 2.0.

[17] V. Cerf. ASCII format for network interchange. RFC 20, Oct. 1969.

[18] Committee IT-014. Australian Technical Specification – E-health XML secured payload profiles, March 2010.

[19] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard), May 2008.

[20] Danske Bank / Sampo Pankki. Encryption, Signing and Compression in Financial Web Services, May 2010. Version 2.4.1.

[21] J. P. Degabriele, A. Lehmann, K. G. Paterson, N. P. Smart, and M. Strefler. On the joint security of encryption and signature in EMV. In O. Dunkelman, editor, *CT-RSA*, volume 7178 of *Lecture Notes in Computer Science*, pages 116–135. Springer, 2012.

[22] J. P. Degabriele and K. G. Paterson. Attacking the IPsec standards in encryption-only configurations. In *2007 IEEE Symposium on Security and Privacy*, pages 335–349. IEEE Computer Society Press, May 2007.

[23] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346 (Proposed Standard), Apr. 2006. Obsoleted by RFC 5246, updated by RFCs 4366, 4680, 4681, 5746.

[24] T. Duong and J. Rizzo. Cryptography in the web: The case of cryptographic design flaws in asp.net. In *2011 IEEE Symposium on Security and Privacy*, pages 481–489. IEEE Computer Society Press, May 2011.

[25] T. Duong and J. Rizzo. Here come the $\oplus$ Ninjas. Unpublished manuscript, 2011.

[26] M. Dworkin. Recommendation for block cipher modes of operation: Galois/counter mode (GCM) and GMAC. In *NIST Special Publication 800-38D, November 2007, National Institute of Standards and Technology. Available at http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf*, 2007.

[27] D. Eastlake, J. Reagle, F. Hirsch, T. Roessler, T. Imamura, B. Dillaway, E. Simon, K. Yiu, and M. Nyström. XML Encryption Syntax and Processing 1.1. *W3C Candidate Recommendation*, 2012. `http://www.w3.org/TR/2012/WD-xmlenc-core1-20121018`.

[28] D. Eastlake, J. Reagle, T. Imamura, B. Dillaway, and E. Simon. XML Encryption Syntax and Processing. *W3C Recommendation*, 2002.

[29] D. Eastlake, J. Reagle, D. Solo, F. Hirsch, and T. Roessler. XML Signature Syntax and Processing (Second Edition). *W3C Recommendation*, 2008.

[30] E. Fujisaki, T. Okamoto, D. Pointcheval, and J. Stern. RSA-OAEP is secure under the RSA assumption. *Journal of Cryptology*, 17(2):81–104, Mar. 2004.

[31] Fuse services framework documentation. Providing Encryption Keys and Signing Keys, July 2012. `http://fusesource.com/docs/framework/2.4/security/MsgProtect-SOAP-ProvideKeys.html#MsgProtect-SOAP-ProvideKeys-SpringConfig`.

[32] D. Gligoroski, S. Andova, and S. J. Knapskog. On the importance of the key separation principle for different modes of operation. In L. Chen, Y. Mu, and W. Susilo, editors, *ISPEC*, volume 4991 of *Lecture Notes in Computer Science*, pages 404–418. Springer, 2008.

[33] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, and H. F. Nielsen. SOAP Version 1.2 Part 1: Messaging Framework. *W3C Recommendation*, 2003.

[34] M. Horsch and M. Stopczynski. The German eCard-Strategy, 2011. Technical Report.

[35] IBM. WebSphere DataPower SOA Appliances. `http://www-01.ibm.com/software/integration/datapower`.

[36] P. Identity. PingFederate. `https://www.pingidentity.com`.

[37] T. Jager, S. Schinzel, and J. Somorovsky. Bleichenbacher's attack strikes again: breaking PKCS#1 v1.5 in XML Encryption. In S. Foresti and M. Yung, editors, *Computer Security - ESORICS 2012 - 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-14, 2012. Proceedings*, LNCS. Springer, 2012.

[38] T. Jager and J. Somorovsky. How to break XML encryption. In Y. Chen, G. Danezis, and V. Shmatikov, editors, *ACM CCS 11: 18th Conference on Computer and Communications Security*, pages 413–422. ACM Press, Oct. 2011.

[39] JBoss Community. JBoss Projects. `http://www.jboss.org/projects`.

[40] M. Jones, J. Bradley, and N. Sakimura. JSON Web Signature (JWS) – draft-ietf-jose-json-web-signature-06, October 2012. `http://tools.ietf.org/html/draft-ietf-jose-json-web-signature-06`.

[41] M. Jones, E. Rescorla, and J. Hildebrand. JSON Web Encryption (JWE) – draft-ietf-jose-json-web-encryption-06, October 2012. `http://tools.ietf.org/html/draft-ietf-jose-json-web-encryption-06`.

[42] J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447 (Informational), Feb. 2003.

[43] B. Kaliski. PKCS #1: RSA Encryption Version 1.5. RFC 2313 (Informational), Mar. 1998. Obsoleted by RFC 2437.

[44] B. Kaliski and J. Staddon. PKCS #1: RSA Cryptography Specifications Version 2.0. RFC 2437 (Informational), Oct. 1998. Obsoleted by RFC 3447.

[45] B. S. Kaliski Jr. On hash function firewalls in signature schemes. In B. Preneel, editor, *Topics in Cryptology – CT-RSA 2002*, volume 2271 of *Lecture Notes in Computer Science*, pages 1–16. Springer, Feb. 2002.

[46] Kantara Initiative. Kantara Initiative eGovernment Implementation Profile of SAML V2.0, June 2010. Version 2.0.

[47] J. Kelsey, B. Schneier, and D. Wagner. Protocol interactions and the chosen protocol attack. In B. Christianson, B. Crispo, T. M. A. Lomas, and M. Roe, editors, *Security Protocols Workshop*, volume 1361 of *Lecture Notes in Computer Science*, pages 91–104. Springer, 1997.

[48] K. Lawrence and C. Kaler. WS-SecurityPolicy 1.2. OASIS Standard, July 2007.

[49] Layer7 Technologies. Layer7 XML Firewall. `http://www.layer7tech.com/products/xml-firewall`.

[50] F. McCabe, D. Booth, C. Ferris, D. Orchard, M. Champion, E. Newcomer, and H. Haas. Web services architecture. W3C note, W3C, Feb. 2004. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

[51] D. A. McGrew and J. Viega. The security and performance of the Galois/counter mode (GCM) of operation. In A. Canteaut and K. Viswanathan, editors, *Progress in Cryptology - INDOCRYPT 2004: 5th International Conference in Cryptology in India*, volume 3348 of *Lecture Notes in Computer Science*, pages 343–355. Springer, Dec. 2004.

[52] Nimbus Directory Services. Nimbus JSON Web Token, May 2012. `https://bitbucket.org/nimbusds/nimbus-jwt`.

[53] NIST. Cipher block chaining. NIST FIPS PUB 81, U.S. Department of Commerce, 1980.

[54] NIST. AES key wrap specification, 2001.

[55] NIST. Recommendation for block cipher modes of operation. Special Publication 800-38A, 2001.

[56] NIST. Recommendation for block cipher modes of operation: The CCM mode for authentication and confidentiality. Special Publication 800-38C, 2004.

[57] Oracle. Securing SOA and Web Services with Oracle Enterprise Gateway, April 2011. Technical Report.

[58] K. G. Paterson, J. C. N. Schuldt, M. Stam, and S. Thomson. On the joint security of encryption and signature, revisited. In D. H. Lee and X. Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, volume 7073 of *Lecture Notes in Computer Science*, pages 161–178. Springer, Dec. 2011.

[59] K. G. Paterson and A. K. L. Yau. Cryptography in theory and practice: The case of encryption in IPsec. In S. Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 12–29. Springer, May / June 2006.

[60] Ping Identity product documentation. PingFederate 6.6, Selecting a Decryption Key (SAML), July 2012. `http://documentation.pingidentity.com/display/PF66/Selecting+a+Decryption+Key+(SAML)`.

[61] J. Rizzo and T. Duong. Practical padding oracle attacks. In *Proceedings of the 4th USENIX conference on Offensive technologies*, WOOT'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[62] P. Rogaway. Problems with proposed IP cryptography. Unpublished manuscript, 1995. `http://www.cs.ucdavis.edu/~rogaway/papers/draft-rogaway-ipsec-comments-00.txt`.

[63] P. Rogaway, M. Bellare, J. Black, and T. Krovetz. OCB: A block-cipher mode of operation for efficient authenticated encryption. In *ACM CCS 01: 8th Conference on Computer and Communications Security*, pages 196–205. ACM Press, Nov. 2001.

[64] SAP. SAP Netweaver. `http://scn.sap.com/community/netweaver`.

[65] J. Schaad, B. Kaliski, and R. Housley. Additional Algorithms and Identifiers for RSA Cryptography for use in the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 4055 (Proposed Standard), June 2005. Updated by RFC 5756.

[66] Shibboleth Consortium. Shibboleth. `http://shibboleth.net`.

[67] J. Somorovsky and J. Schwenk. Technical Analysis of Countermeasures against Attack on XML Encryption – or – Just Another Motivation for Authenticated Encryption. In *SERVICES Workshop on Security and Privacy Engineering*, June 2012.

[68] S. Vaudenay. Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS ... In L. R. Knudsen, editor, *Advances in Cryptology – EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 534–546. Springer, Apr. / May 2002.

[69] K. U. Veiko Sinivee. Encrypted DigiDoc Format Specification, June 2012. Version 1.1.

[70] J. Viega and D. McGrew. The Use of Galois/Counter Mode (GCM) in IPsec Encapsulating Security Payload (ESP). RFC 4106 (Proposed Standard), June 2005.

[71] D. Wagner and B. Schneier. Analysis of the SSL 3.0 protocol. In *Proceedings of the 2nd conference on Proceedings of the Second USENIX Workshop on Electronic Commerce - Volume 2*, WOEC'96, pages 4–4, Berkeley, CA, USA, 1996. USENIX Association.

[72] F. Yergeau. UTF-8, a transformation format of Unicode and ISO 10646. RFC 2044 (Informational), Oct. 1996. Obsoleted by RFC 2279.