



Security Review of RSA Blind Signatures with Public Metadata

Google

Version 1.1 – December 8, 2023

©2023 – NCC Group

Prepared by NCC Group Security Services, Inc. for Google LLC. Portions of this document and the templates used in its production are the property of NCC Group and cannot be copied (in full or in part) without NCC Group's permission.

While precautions have been taken in the preparation of this document, NCC Group the publisher, and the author(s) assume no responsibility for errors, omissions, or for damages resulting from the use of the information contained herein. Use of NCC Group's services does not guarantee the security of a system, or that computer intrusions will not occur.

Prepared By
Parnian Alimi
Giacomo Pope
Thomas Pornin

Prepared For
Google LLC

1 Executive Summary

Synopsis

During the Autumn of 2023, Google engaged NCC Group to conduct a security assessment of the white paper entitled “RSA Blind Signatures with Public Metadata”, along with the corresponding IETF draft for “Partially Blind RSA Signatures”. The work is inspired by the growing importance of anonymous tokens for the privacy of real-world applications. In particular, the paper aims to modify the standard RSA Blind signature protocol such that signatures can only be generated for a specific choice of *public* metadata.

The security assessment of the protocol was performed through an analysis of both the whitepaper and online draft, with direct communication with the Google team. Additionally, a SageMath implementation of the protocol was written following the specification outlined in the IETF draft. The review was performed by three consultants over two weeks for a total of fifteen person-days.

In early November 2023, a retest was performed by two consultants, for four person-days of additional efforts.

Scope

NCC Group’s evaluation included:

- “RSA Blind Signatures with Public Metadata”, a whitepaper available as an *ePrint*.¹
- “Partially Blind RSA Signatures”, an IETF draft specification available online.²

For the November 2023 retest, working versions of the two documents above were provided to NCC Group, with the changes highlighted. These working versions are expected to be published on their respective channels in the near future.

Limitations

NCC Group encountered no significant limitations throughout the project.

Key Findings

During the assessment, a total of two (2) security findings were disclosed, both of low severity:

- finding “Gap in Security Proof of Theorem 2”,
- finding “PSS Encoding Uses an Invalid Length Parameter”.

Additionally, one (1) informational finding was filed.

Retest: all these findings were found to have been fixed.

NCC Group also captured several observations that did not warrant findings, but will be of interest to the Google team, which are contained in the section [Audit Notes](#).

Strategic Recommendations

- Ensure that the reference implementation of the protocol closely follows the specification. For example, the off-by-one error in the PSS padding was obscured by BoringSSL handling the whole operation correctly in test code provided.
- Consider including more detailed comments about potential side-channels and how to mitigate attacks in the specification to aid future implementers.

1. <https://eprint.iacr.org/archive/2023/1199/20230808:012307>

2. <https://datatracker.ietf.org/doc/html/draft-amjad-cfrg-partially-blind-rsa-01>



2 Dashboard

Target Data

Name RSA Blind Signatures with Public Metadata

Engagement Data

Type Protocol Review

Dates 2023-09-18 to 2023-11-06

Consultants 3

Level of Effort 19

Targets

RSA Blind Signatures with Public Metadata

Eprint Paper 2023/1199

draft-amjad-cfrg-partially-blind-rsa-01


IETF Draft for Partially Blind RSA Signatures


Finding Breakdown

Critical issues 0

High issues 0

Medium issues 0

Low issues 2 

Informational issues 1 


Total issues 3

Category Breakdown

Cryptography 3 

Component Breakdown

draft specification 2 

ePrint paper 1 

 Critical

 High

 Medium

 Low

 Informational



3 Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors.

Title	Status	ID	Risk
Gap in Security Proof of Theorem 2	Fixed	NB7	Low
PSS Encoding Uses an Invalid Length Parameter	Fixed	L69	Low
The Key Pair Derivation May Produce Public Exponents Out of the Expected Range	Fixed	7QU	Info



4 Finding Details

Low

Gap in Security Proof of Theorem 2

Overall Risk	Low	Finding ID	NCC-E008730-NB7
Impact	Low	Component	ePrint paper
Exploitability	Undetermined	Category	Cryptography
		Status	Fixed

Impact

The gap in the proof of Theorem 2 may imply a misestimation of the tightness bounds and thus of the actual cryptanalytic resistance of the RSA signature with public metadata protocol. It has no impact on the security proof of the RSA *blind* signatures with public metadata protocol.

Description

In the “[RSA Blind Signatures with Public Metadata](#)” paper, in section 4.1 (page 11), the *Multi-Exponent RSA Assumption* is defined, and Theorem 2 links its security to the more common *RSA Assumption* (with strong modulus). In the standard RSA assumption, the attacker is provided with an RSA modulus N , public exponent e , and target value X modulo N (X is also invertible modulo N), and is challenged with finding Y such that $Y^e = X \pmod N$. In the multi-exponent RSA assumption, the attacker is provided with several public exponents e_i , and must find a value Y that matches the target X for one of the provided exponents e_i . An attacker who can break the standard RSA assumption can obviously break the multi-exponent assumption by simply discarding all the provided exponents except one. Theorem 2 purports to quantify the security in the other direction, i.e. to prove that an attacker who can break the multi-exponent RSA assumption can also break the standard RSA assumption, with a specific bound on the difference of the success probabilities (the *tightness* of the proof). The offered proof of Theorem 2 does so by describing a reduction mechanism, in which an attacker on the standard RSA assumption is built as a wrapper around an attacker on the multi-exponent RSA assumption:

- The attacker A on the standard RSA assumption is presented with a modulus N , public exponent e and target X . The exponent e is an odd integer which is obtained from a relatively small range $[3..e_{\max}]$ (Note: in the paper, the range starts at 1, not 3; but public exponent 1 is trivially broken and should be excluded).
- The attacker can invoke another attacker A' which is assumed to break the multi-exponent RSA assumption, with up to ℓ exponents (e_i) selected in the range $[3..e'_{\max}]$, for some upper bound e'_{\max} which is substantially larger than e_{\max} . The proof tightness depends on the ratio between e'_{\max} and e_{\max} .
- Attacker A then generates ℓ random integers z_i in the $[3..e'_{\max}/e_{\max}]$ range, and computes values $e_i = z_i e$. These values are all in the $[3..e'_{\max}]$ range, so they are valid exponents for the multi-exponent RSA assumption, which A' can break. Thus, they are submitted to A' (along with the modulus N and the target X). A' responds with Y , and an exponent f (which is one of the e_i), such that $Y^f = X \pmod N$. In that case, $f = ze$ for some integer z , and attacker A can compute $Y^z \pmod N$, which is then a valid response to the initial challenge submitted to A .

This reduction is a valid proof only if it can be shown that the sub-challenge sent to A' is indistinguishable from a “real” challenge, i.e. such that the synthetic exponents e_i computed above are indistinguishable from exponents chosen randomly from $[3..e'_{\max}]$. This indistinguishability is necessary, because the success of A' is quantified relatively to the probability



distribution of the e_i values (in the actual RSA blinding scheme, the exponents are chosen from a hash output with uniform distribution among odd integers in the $[3..e'_{\max}]$ range). The paper examines the fact that in the reduction above, all e_i are at least as large as the initial e , and there will be none in the $[1..e-1]$ range; Since e is itself chosen in the $[3..e'_{\max}]$ range, the improbability of such a choice of e_i can be bounded, depending on the ratio e'_{\max}/e_{\max} , which is where that value intervenes in the final tightness.

The proof gap is that the actual selection range of the e_i is not the only way through which the synthetic values may be distinguished from the distribution over $[3..e'_{\max}]$ that the attacker A' expects. The paper simply asserts that the e_i are “random”, but this is not entirely true. Indeed, all the provided e_i are multiples of the same integer e (which is at least 3); this is a very improbable case for randomly chosen integers, especially when ℓ is large (e.g. if $e = 3$, then the probability that ℓ randomly chosen odd integers in a given range are all multiple of 3 is about $3^{-\ell}$, i.e. negligible if $\ell > 80$). Attacker A' can certainly notice the fact that all e_i have a common non-trivial factor, by computing their GCD. In more practical terms, if A' happens to be able to solve the multi-exponent RSA assumption but only if the set of exponents includes several integers that are co-prime to each other, then such an A' would still break the multi-exponent RSA assumption with non-negligible probability, but would not be usable in the reduction (presented in the proof) that tries to break the standard RSA assumption.

It is unclear how the proof can be repaired to close that gap, and what would be the consequences on the proof tightness. The two following points shall be noted:

- Within the paper, Theorem 2 is used in the proof of Theorem 4 (section 5.3) to reduce the unforgeability of RSA signatures with public metadata to the standard RSA assumption. *A contrario*, the unforgeability of RSA *blinded* signatures with public metadata is reduced (in section 6.2) to the chosen-target restricted-exponent RSA inversion assumption, which is a different hypothesis, and that reduction does not use Theorem 2. Therefore, the proof gap shown above has no impact on the blinded signatures.
- None of this constitutes an actual vulnerability: this is not a method to actually break the multi-exponent RSA assumption; this only highlights that it is not yet proven that the multi-exponent RSA assumption is (mostly) as strong as the standard RSA assumption, though this seems intuitively reasonable.

Location

[ePrint paper](#), section 4.1 (page 11)

Retest Results

2023-11-02 – Fixed

The updated paper modifies the proof by embedding the target exponent e into a set of ℓ exponents, the other $\ell-1$ being chosen randomly and uniformly as odd integers in the same range as e . The attacker A' cannot distinguish which of the target exponents is e , and thus breaks it with probability $1/\ell$. The extra ℓ factor is then applied to the tightness of the proof of theorem 4.

Moreover, in annex C, a modified protocol is presented, which generates public exponents with a distribution that matches the old proof for theorem 2, which is included in section C.2 (under the name “theorem 8”). This modified protocol is a bit more complex (and, presumably, doubles the cost of public key operations) and is included in the paper only for the completeness of discourse. In any case, the security analysis of the *blinded* RSA signatures with public metadata does not depend on theorems 2 and 4.



PSS Encoding Uses an Invalid Length Parameter

Overall Risk	Low	Finding ID	NCC-E008730-L69
Impact	Low	Component	draft specification
Exploitability	Undetermined	Category	Cryptography
		Status	Fixed

Impact

The `Blind()` algorithm, as specified, will result in protocol failures half of the time. Such failures are normally detected during `Finalize()`.

Description

The `Blind()` algorithm (described in [section 4.2](#) of the draft specification) applies the PSS padding by invoking the `EMSA-PSS-ENCODE()` function from PKCS#1 ([RFC 8017, section 9.1.1](#)). The bit length of the modulus is provided as second parameter to `EMSA-PSS-ENCODE()`:

```
1. msg_prime = concat("msg", int_to_bytes(len(info), 4), info, msg)
2. encoded_msg = EMSA-PSS-ENCODE(msg_prime, bit_len(n))
   with Hash, MGF, and salt_len as defined in the parameters
3. If EMSA-PSS-ENCODE raises an error, raise the error and stop
```

The `EMSA-PSS-ENCODE()` returns a mostly pseudo-random sequence of bits whose length is exactly the one provided as parameter. In particular, the most significant bit of that sequence (when interpreted as an integer) may be equal to 1 with probability about 1/2.

The *verification* of a signature is delegated to standard RSA, by calling the `RSASSA-PSS-VERIFY()` function; in the blind signatures with public metadata protocol, this is normally done by the requesting client as part of the `Finalize()` operation:

```
6. pk_derived = DerivePublicKey(pk, info)
7. result = RSASSA-PSS-VERIFY(pk_derived, msg_prime, sig) with
   Hash, MGF, and salt_len as defined in the parameters
8. If result = "valid signature", output sig, else
   raise "invalid signature" and stop
```

The `RSASSA-PSS-VERIFY()` function is defined in PKCS#1 ([RFC 8017, section 8.1.2](#)), and, in particular, it invokes `EMSA-PSS-VERIFY()` (the dual function of `EMSA-PSS-ENCODE()`) with a length parameter equal to *one bit less* than the modulus size:

```
3. EMSA-PSS verification: Apply the EMSA-PSS verification
   operation (Section 9.1.2) to the message M and the encoded
   message EM to determine whether they are consistent:
```

```
Result = EMSA-PSS-VERIFY (M, EM, modBits - 1).
```

The end result is that if the most significant bit of the output of `EMSA-PSS-ENCODE()` happens to be equal to 1, then the verification in `EMSA-PSS-VERIFY()` will fail (specifically, in [step 6 of that algorithm](#)). Indeed, among the four test vectors provided in the draft specification (appendix B), this happens with test vectors 2 and 3; however, test vectors 1 and 4 do not trigger the issue because `EMSA-PSS-ENCODE()` happens to return values whose most significant bit is zero in these two test vectors.



Note: This issue appears to have been imported from the [draft specification on blind RSA signatures](#) (without public metadata), which has the same issue in its own `Blind()` algorithm. The [test code](#) linked from the ePrint paper avoids the issue because it does not call `EMSA-PSS-ENCODE()` directly; it calls the BoringSSL function `RSA_padding_add_PKCS1_PSS_mgf1()`, providing the public key itself and letting the function extract and use the correct length from it.

Recommendation

Replace the value `bit_len(n)` with `bit_len(n) - 1`. The same issue should be reported to the authors of the draft specification on RSA blind signatures.

Location

[draft specification, section 4.2](#)

Retest Results

2023-11-02 – Fixed

This issue was fixed as suggested in [commit 7ed9aba](#).



The Key Pair Derivation May Produce Public Exponents Out of the Expected Range

Overall Risk Informational

Impact Low

Exploitability None

Finding ID NCC-E008730-7QU

Component draft specification

Category Cryptography

Status Fixed

Impact

The public exponents derived from the public metadata may exceed the range analyzed in the ePrint paper.

Description

In the [ePrint paper](#), public exponents are derived from the public metadata, through a pseudorandom function that outputs odd integers of length at most $\lambda-3$ bits, with λ being the size (in bits) of the modulus prime factors. It is shown that since the modulus prime factors are strong primes, then such a public exponent is necessarily valid, i.e. invertible modulo $\varphi(n)$. The paper, in fact, underestimates the bound by 1 bit, and public exponents may range up to $\lambda-2$ bits in size while still guaranteeing invertibility.

In [section 4.6](#) of the draft specification, this derivation process is described with the `DerivePublicKey()` function:

```
3. lambda_len = modulus_len / 2
4. hkdf_len = lambda_len + 16
5. expanded_bytes = HKDF(IKM=hkdf_input, salt=hkdf_salt, info="PBRSA", L=hkdf_len)
6. expanded_bytes[0] &= 0x3F // Clear two-most top bits
7. expanded_bytes[lambda_len-1] |= 0x01 // Set bottom-most bit
8. e' = bytes_to_int(slice(expanded_bytes, lambda_len))
```

`modulus_len` is the length of the modulus, expressed in bytes. The `DerivePublicKey()` function generates `lambda_len` bytes, then clears the two most significant bits (and sets the least significant), to get an odd integer in, presumably, the expected range. However, since the lengths are here expressed in bytes and not in bits, rounding may lead to generating public exponents larger than expected. For instance, if using an RSA public key of size 2500 bits, with two prime factors of length 1250 bits each, then `modulus_len` is equal to 313, `lambda_len` is 157, and the generated `e'` will have a length up to 1254 bits, i.e. larger than the expected 1248 bits.

This issue is unlikely to lead to adverse consequences, since most RSA public keys have a size which is a multiple of a large power of 2 (e.g. sizes of 2048 or 3072 bits are common), in which case the sizes in bytes are as precise as the sizes in bits, and the range specified in the paper is respected. Moreover, even if the public exponent is larger than $\lambda-2$ bits, the probability that the generated value turns out not to be invertible modulo $\varphi(n)$ is negligible.

Recommendation

To better align the specified protocol with the mathematical analysis, compute the actual value of λ from the length of the modulus, expressed in bits, to avoid rounding effects.



Retest Results

2023-11-02 – Fixed

[Commit 9741fdc](#) adds the requirement to the `DerivePublicKey()` function that the modulus length, in bytes, “MUST be a power of 2”. With this restriction, there are no rounding effects and the computed public exponents match the range specified in the ePrint paper. This fixes this issue.



5 Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Rating	Description
Critical	Implies an immediate, easily accessible threat of total compromise.
High	Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.
Medium	A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.
Low	Implies a relatively minor threat to the application.
Informational	No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

Rating	Description
High	Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.
Medium	Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.
Low	Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

Rating	Description
High	Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.



Rating	Description
Medium	Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.
Low	Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Category Name	Description
Access Controls	Related to authorization of users, and assessment of rights.
Auditing and Logging	Related to auditing of actions, or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to mathematical protections for data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing system failure.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Timing	Related to race conditions, locking, or order of operations.



6 Audit Notes

This section contains some notes about the paper and the protocol. None of these remarks constitutes a security issue.

Retest (November 2023): the paper and protocol specification authors have taken all the following remarks into account in their working documents, following NCC Group's recommendations or with other changes yielding equivalent results.

ePrint Paper

This subsection relates to the “[RSA Blind Signatures with Public Metadata](#)” paper on ePrint, in its version from August 10th, 2023.

Private Exponent Computation

In various places, starting on page 3, it is asserted that the RSA private exponent d is an integer such that $ed = 1 \pmod{\varphi(N)}$. This is not strictly necessary: the RSA primitive works as long as d is inverse of e modulo $p-1$ and $q-1$, i.e. modulo the least common multiple of $p-1$ and $q-1$; the latter can be computed using Carmichael's function and results in a smaller integer which divides $\varphi(N)$. Indeed, with $p = 2p'+1$ and $q = 2q'+1$, $\varphi(N) = 4p'q'$ but it suffices that d is an inverse of e modulo $2p'q'$. This changes nothing to the analysis in the paper.

More importantly, practical RSA implementations would not bother to compute d at all (except possibly for compliance with the standard RSA private key storage format specified in [PKCS#1](#)). Indeed, efficient RSA implementations do not perform the exponentiation modulo N , but modulo p and modulo q , separately, reassembling the two results with the [Chinese remainder theorem](#). Since exponentiation complexity (for typical RSA-sized values) is cubic in the size of the modulus, using the CRT yields a performance speed-up by factor of close to 4. In that context, an implementation does not compute d itself, but d modulo $p-1$ and modulo $q-1$, separately.

The inversion of e modulo $p-1$ can furthermore be done efficiently (and securely, in particular without leaking information through side-channels) with an [optimized binary GCD](#) modulo $p' = (p-1)/2$ (working modulo p' is here necessary since the binary GCD requires an odd modulus). Once the inverse modulo p' is performed, it can be adjusted to become an inverse modulo p by conditionally adding p' to the inverse of e modulo p' (specifically, the final inverse modulo $p-1$ must be an odd integer, so the adjustment consists in adding p' if and only if the inverse of e modulo p' happens to be an even integer). Since the binary GCD cost is mostly quadratic in the size of the modulus, its cost should be mostly negligible with regard to the exponentiation cost of the signing operation itself. The paper attributes (end of section 7, page 25) the signature overhead in the signature with public metadata (compared with the “plain RSA” signatures) to the cost of this inversion, but with such an optimized inversion routine, that overhead should be minimal.

Public Exponent Range

In several places, starting in section 3.1 (page 8), the paper asserts that since p has length λ bits, p' has length at least $\lambda-2$ bits; from this assertion is obtained the rule that the exponent should be chosen to have length at most $\lambda-3$ bits (and γ in Lemma 1 page 17 is chosen of length at most $\lambda-4$ bits). This is a slight underestimate: if p has length λ bits, i.e. $p > 2^{\lambda-1}$, then $p' \geq 2^{\lambda-2}$, i.e. p' has length at least $\lambda-1$ bits, and the exponents can be safely chosen to have length up to $\lambda-2$ bits.

At the other end of the range, in several places in section 4, RSA exponents are assumed to be chosen uniformly among odd integers in the $[1..e_{\max}]$ range for some bound e_{\max} . The value 1 itself should be excluded, since inverting RSA with exponent 1 is trivial! The minimal valid exponent for RSA is 3.



Key Pair Generation Cost

The generation of strong primes is much more expensive than the generation of a random prime (typically by a factor 50 or so), which makes key pair generation for the proposed scheme quite costlier than for plain RSA. This is somewhat elided in the paper, which asserts that “the Setup algorithm is essentially identical to the standard RSA signatures without public metadata protocols”, and presents no measurement on key pair generation in the benchmarks (section 7 and figure 10). Of course, key pair generation is an infrequent operation (server-side only, and typically done on a yearly basis), so the extra cost is unlikely to be an issue in practical deployment.

Footnote about Euler’s Totient function

Footnote two correctly states that Euler’s totient function counts the number of integers smaller and also coprime to an input N . However, in the context given, it may be of more use to the reader to learn that Euler’s totient function effectively computes the order of the multiplicative group of units $(\mathbb{Z}/N\mathbb{Z})^*$. This is particularly of note, as in the next sentence the group of units is introduced using mathematical notational without a concrete explanation for those unfamiliar.

Typographic Errors

- page 1 (abstract, first line): “to provider users” -> “to provide users”
- page 1 (first paragraph): “in the context electronic cash” -> “in the context of electronic cash”
- page 2 (line 2): “an signer” -> “a signer” (twice)
- page 2 (line 3): “pairs of message and valid signatures” -> “... signature”
- page 3 (second paragraph): “check that the public metadata D ” -> “check the public metadata D ”
- page 4 (second paragraph): “before proving security” -> “before proving the security”
- page 7 (end of section 2.1): “may be performed to anyone” -> “may be performed by anyone”
- page 7 (second paragraph of 2.2): “finally the challenge” -> “finally the challenger”
- page 8: “the possible choices of public metadata is not too larger” -> “the set of possible choices [...] is not too large”
- page 16: “that essentially uses” -> “that essentially use”
- page 16 (end of 5.1): “these hash function” -> “... functions”
- page 17: “we must come up a way” -> “... come up with a way”
- page 24: “after augmented” -> “after augmenting”
- page 24: “the scenarios studied in [25] is more favorable” -> “... are more favorable”
- page 25: “unlinkabiity” -> “unlinkability”

IETF Draft

This subsection relates to the IETF Draft “Partially Blind RSA Signatures”, in its version `draft-amjad-cfrg-partially-blind-rsa-01`. An automatically generated up-to-date version of the draft specification is also [available](#); at the time of writing this report, that version differed from the 01 draft only in inconsequential ways (q is renamed to p' in the `SafePrime()` function, and the reference to the draft specification for blind RSA signatures (without metadata) is updated from version 13 to 14). The notes below are ordered by the draft specification sections.



Section 1

At the end of section 1, it is asserted that the resulting signatures “can be verified with a standard RSA-PSS library”. This assertion might require some restrictive qualifiers, because not all standard RSA-PSS library can actually verify them. For instance, [FIPS 186-4](#) mandates that the public key size shall be only 1024, 2048 or 3072 bits; a FIPS 186-4-compliant library could legitimately reject signatures of any other size. Moreover, FIPS 186-4 is about to be officially withdrawn, and is superseded by [FIPS 186-5](#); in FIPS 186-5, modulus sizes of 2048 bits or more are supported (provided that they are even), but public exponents are restricted to the 2^{16} to 2^{256} range. The public exponents obtained from `DerivePublicKeys()` are larger than 256 bits, and thus not formally supported by FIPS 186-5.

From a more practical point of view, one major cryptographic library is [SymCrypt](#); it is the library that implements the cryptographic operations in the Windows operating system. This library supports public RSA exponents [only up to 64 bits](#).

Section 3

The text should probably state somewhere whether the `random_prime()` and `is_prime()` functions may use probabilistic primality tests, or instead non-probabilistic proofs of primality are required. FIPS 186-5 (appendix A.1.1), for instance, makes the distinction between “provable primes” and “probable primes”, and mandates different ranges for these cases.

Section 4.1

The section does not mandate any specific allowed range for the generated RSA keys. For security, keys of at least 2048 bits are recommended; an explicit upper bound may also be specified, since very large keys are expensive to support and can make signature verification engines susceptible to denial-of-service attacks.

The RSA private key is specified as `(n, p, q, phi, d)`. This departs from PKCS#1 ([RFC 8017, section 3.2](#)), which defines an RSA private key as being either `(n, d)`, or `(p, q, dP, dQ, qInv)`. RFC 8017 also defines ([appendix A.1.2](#)) an ASN.1-based encoding format for private keys that includes `(n, e, d, p, q, dP, dQ, qInv)`. The values `d` (full-size private exponent) and `phi` (Euler’s totient for the modulus) are not actually needed for private key usage, when `p` and `q` are provided, which is the recommended case since it allows for much faster operations (by a factor of about 4). For the blind RSA signatures with public metadata, `d` and `phi` are not needed either, since `dP` and `dQ` can be computed as the inverses of the public exponent `e` (or `e'` in case of public key derivation) modulo `p-1` and `q-1`, respectively (computing such inverses is also more efficient than doing it modulo `phi`, which is a twice larger integer). Expressing private keys with `(p, q, dP, dQ, qInv)` would be better aligned with PKCS#1 and would promote computing efficiency.

The `KeyGen()` function is specified as taking as parameter a target modulus size in bits (an even integer), then generates the two prime factors with half that size:

1. `p = SafePrime(bits / 2)`
2. `q = SafePrime(bits / 2)`

This process can result (with probability about 1/2) in a modulus which is one bit shorter than the requested size; for instance, a 2048-bit modulus key is expected, but a 2047-bit key is obtained. This can induce interoperability issues, since some standards (in particular FIPS 186-5) mandate use of even-sized moduli only. To avoid this issue, candidate primes for bit length k should be generated in a range whose minimum bound is larger than $2^{k-0.5}$. A simple way, used in some libraries (e.g. OpenSSL) is to forcibly set to one the two most significant bits of the prime candidate.



The `SafePrime()` process, as presented, is very expensive. A typical primality test is the combination of two methods, first a fast one (such as trial divisions) to eliminate most bad candidates, then an expensive one (Miller-Rabin) to ensure primality with a high enough probability. To put rough numbers on it, over 1000 candidates, one will be prime; the fast method will eliminate all but about 50 of the non-primes. Thus, generating a safe prime with the specified method will require about $50 \times 1000 = 50000$ calls to the expensive primality test (50 times for each “inner prime” p' , to be done 1000 times until the outer value $2 \cdot p' + 1$ is prime too. A much faster method would run the fast primality test on *both* the inner (p') and outer (p) candidates before trying the expensive test on either value. This would reduce the number of invocations of the expensive primality test to $50 \times 50 = 2500$, i.e. a reduction of the cost by a factor of 20. These numbers are only approximations, and the actual gain may vary, but the speed-up is certainly not negligible. In its current presentation, the draft specification discourages use of an alternative key generation algorithm, since in the introduction of section 4, the use of the specifically described algorithm is “REQUIRED”.

Section 4.2

The text says that if a “blinding error” occurs, then “implementations SHOULD retry the function again”. For a properly formed public key, the probability of randomly hitting a value which is not co-prime with the modulus is truly negligible – it is many orders of magnitude lower than that of a hardware error. Some system malfunctions are transient and retrying might cure them, but usually such decisions are better handled at a higher application level. Apart from hardware-related failures, a blinding error would indicate an invalid public modulus, possibly maliciously crafted or altered in transit, and in such cases, retrying the function is arguably not the right behaviour, since it can only help with attacks (e.g. automatically repeated computations are helpful for leveraging side-channel attacks).

Furthermore, there are two checks of co-primality with the modulus in the `Blind()` function, in step 6 (for the encoded message) and in step 9 (for the blinding value). In the common case of the encoded message being randomized (either through the message preparation, with `PrepareRandomize()`, or through the PSS encoding, which uses a random salt), both checks would happen with the same probability under the same conditions (invalid modulus or hardware error), and should be handled similarly, i.e. the function should be retried in both cases, or, preferably, *not* retried in either case. The calling application is better placed to make decisions about the relevance (or even logical possibility) of “updating the public key” or of trying again in case faulty hardware is suspected.

Algorithm `Blind()` is defined as taking as parameters *both* the public key, and the length of the modulus (`modulus_len`). This seems redundant, and no check is performed to ensure that both values match. In a practical implementation, the modulus length should not be provided separately, but instead obtained from the modulus itself, which is part of the public key. The same remark applies to algorithms `BlindSign()` (section 4.3), `Finalize()` (section 4.4), `DerivePublicKey()` (section 4.6) and `DeriveKeyPair()` (section 4.7).

The text hints at the requirement of uniform selection for the blinding value r , but provides no guidance except the expression “rejection sampling”. A possible reference here is appendix A.3.2 of FIPS 186-5, which describes rejection sampling for uniform selection of an integer modulo another integer (in the context of ECDSA signatures, but the methodology applies generically).

Section 4.3

In algorithm `BlindSign()`, the core RSA exponentiation is performed in step 3 (`RSASP1()` call), then the signer verifies that the output is correct in steps 4 and 5 (`RSVP1()` call). Mathematically, the verification is entirely redundant: it *cannot* fail, if the server’s key is not malformed. This is true even if the input m happens not to be co-prime to the modulus (the



RSA operation is a permutation of the entire set of integers modulo n , not just the invertible integers). A failure here may happen only if the server's own private key is altered, or on hardware failure, so this again raises the question of performing this test, especially since it is expensive (the derived public exponent is about half the size of the modulus, and `RSAP1()` uses only the public modulus, so that step would be about twice more expensive than the `RSASP1()` call itself, which can leverage knowledge of the prime factors p and q).

Section 4.4

In the `Finalize()` algorithm, the blinded signature obtained from the server is verified (at step 1) to have the right length in bytes (i.e. the same length as the modulus, but there is no check that the corresponding decoded integer z is actually lower than the modulus n). This lack of test does not seem to induce any exploitable vulnerability, but it may make the implementation of step 3 (modular multiplication of z with `inv`) more complicated, since that code must then be able to cope with some slightly out of range input values.

Section 4.5

The text discusses the difference between the signed message itself (`msg`) and the *prepared* message (`input_msg`), which is its randomized version (when `PrepareRandomized()` is used). This is somewhat confusing, because the previously presented algorithms (`Blind()` and `Finalize()`) take as input a message called "`msg`", which is *not* the signed message itself, but the prepared message (i.e. `input_msg`). It also keeps somewhat implicit the important fact that when the message is randomized, the 32-byte random prefix must be retained, since it is needed for verification of the signature. The presentation would be clearer if that random 32-byte prefix was made part of the signature (that is, if the signature was in fact the concatenation of the 32-byte random prefix and the encoded big integer modulo n).

Section 4.6

The HKDF function is used, but it is not defined anywhere in the document. A reference to [RFC 5869](#) is needed here.

In `DerivePublicKey()` (step 4), the HKDF output length (`hkdf_len`) is set to `lambda_len + 16`, even though only the first `lambda_len` bytes of the HKDF output are used. The production of 16 extra bytes does not seem to have any usefulness.

Section 5.2

The specification states that "The RECOMMENDED method for generating the server signing key pair is as specified in FIPS 186-4". This is incorrect: the process specified in FIPS 186-4 does not produce the strong primes that are needed for blinded RSA signatures with public metadata. Moreover, this statement is in direct contradiction with the introduction of section 4, that asserts that the use of the key pair generation specified in section 4.1 is "REQUIRED". Section 7.1 reiterates that it is "essential" and "integral" that the standard RSA key pair generation method is *not* used. The sentence that starts section 5.2 is apparently copied from section 6.2 of the blinded RSA signatures draft specification (without public metadata), and was probably meant to be deleted.

Section 6

The name "RSAPBSSA-SHA384-PSS-Deterministic" is derived from the corresponding non-metadata name "RSABSSA-SHA384-PSS-Deterministic"; both are somewhat confusing because they feature the term "deterministic" but still produce randomized signatures.

Section 7

The text refers to "section 8 of [RSABSSA]" for the security analysis; it's actually section 7 of that draft in its current version (14).



The referenced specification asserts that the *-Deterministic variants must not be used if the public key cannot be ascertained through other means to be non-malicious. This is not fully true, in that a sufficiently large PSS salt (at least 16 bytes) provides enough randomization of the encoded message that it maintains unlinkability even if the public modulus is maliciously crafted. Only the variants with a PSS salt small enough to allow adversarial enumeration of all possible salt values, *and* a non-randomized input, will be weak against such malicious signers. This might be worth a mention, because RSAPBSSA-SHA384-PSS-Deterministic is probably the most convenient variant for applications, since it does not require storage and transmission of the random 32-byte message prefix along with the resulting signature.

In sub-section 7.4, a denial-of-service situation is evoked, described as follows:

In particular, an attacker can pick public metadata such that the output of `DerivePublicKey` is very large, leading to more computational cost when verifying signatures.

The output of `DerivePublicKey` is mostly the public exponent, which has a bounded size. Some public exponents are mathematically a bit shorter than others, though this will hardly have any real impact on performance; on average, about half of the public exponents will have the maximum size (which is about half the size of the modulus) and there is no possibility for an attacker to force the verification to use an exponent larger than that. This denial-of-service attack description seems to be a remnant of an older draft version, in which public exponents might have been generated differently and allowed to grow much further.

[Section 7.1 of the blind RSA signatures specification](#) contains a brief passage about side-channel attacks. The server is more naturally threatened by such attacks since it is amenable to repeated requests, allowing attackers to leverage small timing biases through statistical analysis. In the signatures without metadata, the server-side situation is not much different from normal RSA signatures; however, when public metadata are used, the server performs additional operations that involve the private key, namely the derivation of the metadata-specific key pair. The inversion of the derived public exponent modulo $\varphi(N)$ (or modulo $p-1$ and modulo $q-1$) is a potential target for side-channel attacks; the classic extended Euclidean GCD algorithm is not constant-time. Inversion modulo $p-1$ can be performed modulo p' first, then lifted to $p-1 = 2p'$ with a single conditional addition; since p' is odd, a binary GCD can be used, for which [optimized and constant-time implementations](#) exist.

Section 9

Reference [DSS] points to FIPS 186-4, which, as pointed out previously, is officially superseded by FIPS 186-5; FIPS 186-4 will then move to “withdrawn” status in the near future (on February 3rd, 2024), making it unsuitable for reference purposes. The reference should be updated to point to FIPS 186-5.

Reference [RSABSSA] is listed in the “informative references”, but it is used explicitly in some important parts of the protocol, e.g. the message preparation (`PrepareRandomized()` and `PrepareIdentity()`). It should be moved to the “normative references” section.

Appendix B

The appendix states that the test vectors use RSAPBSSA-SHA384-PSS-Randomized; this is not true. The test vectors use RSAPBSSA-SHA384-PSS-Deterministic (no random 32-byte prefix is provided, nor used).



The presentation of the text vectors is somewhat underspecified:

- The numerical values are said to be “encoded as a hexadecimal string”, which is ambiguous. The text should precise that unsigned big-endian convention is used.
- The message blinding value is called `blind` in the test vectors; in the `Blind()` algorithm specification (section 4.2), it was called `r`.
- A `salt` value is provided, but there is no such parameter or variable in the algorithms described in the specification. It is really the value generated randomly within the `EMSA-PSS-ENCODE()` function in RFC 8017, section 9.1.1 (step 4).
- The `blinded_sig` value in the test vectors was called `blind_sig` in the `BlindSign()` and `Finalize()` algorithms (section 4.3 and 4.4).

All the test vectors could be verified with a custom implementation (in Sage), made by following RFC 8017 (for PSS encoding/decoding) and the draft specification, with two modifications: the value `bit_len(n) - 1` was used as second parameter to `EMSA-PSS-ENCODE()` (see [finding "PSS Encoding Uses an Invalid Length Parameter"](#)), and the provided `msg` values were used directly as input to `Blind()` and `Finalize()`, assuming that the variant was really RSAPBSSA-SHA384-PSS-Deterministic. If RSAPBSSA-SHA384-PSS-Randomized test vectors are added, then they must include the random message prefix in some way.

Typographic Errors

- (Section 4.1) ‘hte’ -> ‘the’
- (Section 4.2) ‘an “blinding error” error’ -> ‘a “blinding error” error’ (in the text and in algorithm `Blind`, step 9)
- (Section 5.1) ‘implementors’ -> ‘implementers’
- (Section 6) ‘instantation’ -> ‘instantiation’
- (Section 7.1) ‘the resulting outputs [...] does cause errors’ -> ‘... do cause errors’
- (Section 7.2) ‘the security analysis [...] remain true’ -> ‘... remains true’
- (Section 7.2) ‘The DerivePublicKey’ -> ‘The DerivePublicKey function’
- (Section 7.4) ‘suspectible’ -> ‘susceptible’
- (Appendix B) ‘messsage’ -> ‘message’

