# Rolling Re-evaluation of the Swiss Post e-Voting System: Versions 1.2.3 and 1.3.0

## Audit Scope 1: Cryptographic Protocol

Aleksander Essex

Department of Electrical and Computer Engineering
Western University, Canada
`aessex@uwo.ca`

August 1st, 2023

Submitted to the Swiss Federal Chancellery

## Management Summary

In cooperation with the Chancellery, I re-evaluated the Swiss Post e-voting system covering changes made in versions 1.2.3 and 1.3.0. In total, I identified 12 issues between the two versions (14 issues, of which 2 have been resolved since the initial draft).

As in previous reports, many of these findings pertain to minor issues surrounding ambiguities or typos in the specification. However, we make the following three high-level recommendations.

1. **Improving written communication regarding design intent.** As an ongoing issue, we continue to encourage Swiss Post to explain their reasoning for their changes in writing with particular attention to why (in their view) a given change is necessary and sufficient.

2. **Improving cryptographic parameter generation.** As an ongoing issue, we continue to recommend Swiss Post discontinue the use of the Baillie-PSW primality test. To that end, we include a detailed proposal for a safe prime generation algorithm in Section 5.3 that addresses Post's performance needs while simultaneously providing the security properties outlined in the recommendations of previous reports.

3. **Expanding the discussion on voter authentication.** We articulate concerns about the suitability of voter dates of birth as an authentication credential and recommend a broader-ranging conversation and analysis of the real-world security requirements of voter credentials.

## Version History

| | |
|---|---|
| August 1st, 2023 | Final draft. Issue 6 (Signature Algorithm Specification) and Issue 7 (Signatures on Messages) marked as resolved based on Swiss Post's response to our initial draft. |
| July 2nd, 2023 | Initial draft submitted to Chancellery. |

# Table of Contents

## 1 Description

This document is part of a rolling re-examination of the Swiss Post e-voting system. In this report, we examine system updates made in version 1.2.3 (March 2023), which we discuss starting in Section 1.4, and version 1.3.0 (May 2023), which we discuss starting in Section 3. Finally, as a significant component of this report, we conduct a detailed analysis of Swiss Post's safe prime generation algorithm beginning in Section 5.3 and propose a concrete solution to meet the most important design goals simultaneously.

Between these two versions, I initially identified 14 issues, which are catalogued in the following sections. Of these, 2 issues have been marked as resolved since the initial draft (Issues 6 and 7) on account of Swiss Post's clarifications. The following subsections, however, detail the more important high-level findings.

### 1.1 Improving Written Communication Regarding Design Intent

In several instances in the recent versions, the reasoning and intent behind a particular change were not explained in the document, although it became apparent after a direct conversation with the Swiss Post team.

A concrete example is given in Issue 2 in Section 2. After repeated recommendations in my previous reports to use the Miller-Rabin test instead of the Baillie-PSW test, Swiss Post updated the system—to use *both*. From a cryptographic and software design perspective, this choice did not initially appear to make sense, although a telephone conversation with the Swiss Post designers clarified their intent.

As such, I would continue to encourage Swiss Post not only to improve their system incrementally but also to explain the reasoning for their changes in writing. Particular attention should be given to why (in their view) a given change is necessary and sufficient.

### 1.2 Improving Cryptographic Parameter Generation

We continue to recommend Swiss Post discontinue the use of the Baillie-PSW primality test due to several security and design factors covered in Section 1.4 (and extensively in our previous reports to the Chancellery). Speaking with Swiss Post, we understand that performance is a key issue for them.

To that end, we propose a safe prime generation algorithm all meeting our (combined) design goals simultaneously: formal, deterministic guarantees; faster run time (29% faster than Swiss Post's current approach); and a simpler, self-contained design with no reliance on external primality testing software libraries. Our approach is based on efficient deterministic primality generation based on Pocklington's theorem, similar to FIPS 186-5 [1] (see Appendix B.10) except adapted to the safe prime setting, with an additional safe prime-focused optimization due to a recent observation by Ramzy [9]. A detailed proposal including algorithms, analysis, proofs, code and testing is presented starting in Section 5.3.

### 1.3 Expanding Discussion on Voter Authentication

We articulate concerns about the suitability of voter dates of birth as an authentication credential and recommend a broader-ranging conversation and analysis of the real-world security requirements of voter credentials.

### 1.4 Documents Examined

Below is a list of the versions of my reports submitted to the Chancellery, the version of the primitives specification examined in my report, and the date that version was published by Swiss Post.

**Primitives Specification**

**Description:** Pseudocode specifications of cryptographic functions used by the Swiss Post system. Referred to throughout this document as the *primitives specification*.

| Report | Version Examined | Date Published |
|---|---|---|
| 2021 Preliminary Report | 0.9.5 | 2021-06-22 |
| 2021 Final Report | 0.9.8 | 2021-10-15 |
| 2022 Re-Examination | 1.0.0 | 2022-06-24 |
| 2022 Re-Examination (Addendum I) | 1.0.0 | 2022-06-24 |
| 2023 Addendum II | 1.2.0 | 2022-12-09 |
| 2023 Rolling Re-examination (this document) | 1.2.1 | 2023-06-30 |
| 2023 Rolling Re-examination (this document) | 1.3.0 | 2023-06-30 |

**Available:** https://gitlab.com/swisspost-evoting/crypto-primitives/crypto-primitives/-/blob/master/Crypto-Primitives-Specification.pdf

**System Specification**

**Description:** Document describing the steps, phases and procedures of setting up, executing and verifying an election using the Swiss Post system. Referred to in this document as the *system specification*.

| Report | Version Examined | Date Published |
|---|---|---|
| 2021 Preliminary Report | 0.9.6 | 2021-06-25 |
| 2021 Final Report | 0.9.7 | 2021-10-15 |
| 2022 Re-Examination | 1.0.0 | 2022-06-24 |
| 2023 Rolling Re-examination (this document) | 1.3.0 | 2023-06-30 |

**Available:** https://gitlab.com/swisspost-evoting/documentation/-/blob/master/System/System_Specification.pdf

# Re-Examination of System Version 1.2.3

In Sections 1.4 to 3, we discuss relevant findings in changes made to Swiss Post's e-Voting system as of version 1.2.3. Specifically, we examine the changes made to the cryptographic primitives specification document as of version 1.2.1.[1] We focus on primarily on the relevant changes highlighted in Swiss Post's CHANGELOG.[2]

Swiss Post continues to refine its system in a positive and constructive direction. There are some concerns over some algorithmic changes, which would benefit from additional (written) explanations regarding the particular security goals/requirements. Without articulated security requirements or goals for generating verifiable parameters, assessing the necessity and sufficiency of a specific algorithmic change is difficult.

As a high-level recommendation, the approach to cryptographic design ought to be one of *parsimony*; necessary things must be included. Unnecessary things ought to be excluded. This lowers the threat surface and limits confounding factors in the analysis.

[1] Swiss Post Cryptographic Primitives specification version 1.2.1. Available: https://gitlab.com/swisspost-evoting/crypto-primitives/crypto-primitives/-/blob/745ee18fd65684b685c7adc040efc55220f667b3/Crypto-Primitives-Specification.pdf

[2] Swiss Post Cryptographic Primitives Changelog for version 1.2.1. Available: https://gitlab.com/swisspost-evoting/crypto-primitives/crypto-primitives/-/blob/745ee18fd65684b685c7adc040efc55220f667b3/CHANGELOG.md

## 2 Primality testing and verifiable parameter generation

This section primarily concerns changes made to primality testing (Section 7.1) and parameters generation (Section 7.2).

**Issue 1:** Double-testing of primes

**Description**: `GetEncryptionParameters` in the primitives specification applies Baillie-PSW and Miller-Rabin. Essentially a prime is tested twice using two different methods. The purpose of this was initially unclear. However, speaking to the Swiss Post team directly, their intent seems to have been to initially apply Baillie-PSW to a perceived performance advantage and apply Miller-Rabin as a final step to ensure formal bounds exist on the probability the output $p, q$ are prime.

As we discuss in Section 7.3, except base-2 strong pseudoprimes, there is no performance advantage to BPSW when testing composite integers, which accounts for the overwhelming number of invocations of primality testing. The main speedup comes when $q$ is prime, in which case BPSW runs several times faster than Miller-Rabin. However, at the 3072-bit level, this even occurs less in than 0.1% of candidates, at which point Miller-Rabin is then being applied anyway. Furthermore, if $q$ was determined to be prime, testing BPSW and *then* Miller-Rabin on $p$ is necessarily slower than testing Miller-Rabin alone.

However, as we discuss in Section 9.4, if $q$ is prime, the primality of $p$ can be established without BPSW or Miller-Rabin.

**Recommendation:** The recommendation remains as it has for the past year: Discontinue the use of the BPSW primality test. As we show in Section 10.1, faster methods exist, which also come with formal, deterministic bounds.

**Issue 2:** Unnecessary complexity in primality testing

**Description**: The primality testing description has added additional complexity (Sec 7.1 of primitives spec version 1.2.1). The document now requires *two* two approaches to primality testing: (a) Swiss Post's custom variant of the Baillie-PSW test, but now also (b) a conventional application of Miller-Rabin. Why are both approaches used? What are they contributing?

**Recommendation:** Switching to deterministic generation of safe primes can eliminate reliance on external primality testing algorithms. A detailed proposal, including algorithms, analysis, proofs, code and testing, is presented starting in Section 6.

**Issue 3:** Non-uniform prime generation

**Description**: The modification of `GetEncryptionParameters()` appears to be based on my feedback in Addendum II (January 2023) and appears to have eliminated the domain separation issue. I recommended an approach to domain separation in the hash function that retained the overall functionality.
Swiss Post's new approach, however, diverges from this. The new method no longer invokes the hash on the counter value and tests the integer-ized hash value for primality. Rather, the counter is incremented, sieving for non $B$-smooth values. It is then the counter which is tested for primality. The result is that primes are now no longer selected uniformly in the set of primes $\mathbb{P}$. Rather, an integer is selected uniformly, and then the next highest prime is selected.
This approach introduces a statistical bias due to varying gaps between primes. I ran a basic test at the 1000-bit level and found that approximately 20% of primes account for over 50% of the primes that this algorithm would select. Some primes are more likely to be chosen than others.
This bias may not be of any particular consequence. However, compared to the total cost of the modular exponentiations of the primality testing, the cost of sieving for non $B$-smooth integers seems minor compared to the standard approach of generating a random integer and testing for $B$-smoothness via trial division.

**Recommendation:** Please provide some analysis of this approach, both in terms of the security *cost* (to what degree does uniformity in $\mathbb{P}$ matter), and the performance *benefit* (quantified relative to the standard approach).

**Issue 4:** Non-standard approach to verifiable parameter generation

**Description**: Swiss Post's new approach of increment/test/repeat departs from the NIST standard approach to verifiable parameter generation of hash/test/increment/repeat. See, e.g., Appendix C.3 of [4] or Appendix A.1 of [1].

**Recommendation:** Following with past recommendations, I encourage Swiss Post to provide additional explanation in the primitives specification.

## 3 RecursiveHashToZq

This section examines changes made to the recursive hash (Section 4.2). In particular, I examined their new approach to `RecursiveHashToZq` (Algorithm 4.9), which seeks to minimize modulo bias.

| **Issue 5:** No analysis or definitions of modulo bias mitigation |
|---|
| **Description**: The approach is to output the bytes necessary to reach all possible values in $\mathbb{Z}_q$, *plus* and 256 excess bytes to mitigate the modulo bias, i.e., output $|q| + 256$ bytes. Swiss Post says the modulo bias is now sufficiently, "small," which I don't dispute. However, no analysis is provided.<br>Additionally, they claim the modulo bias is "negligible." The word *negligible* is being used informally here. However, it also has a formal definition in this context. Technically, since 256 is not a function of the security parameter, it's not negligible, at least not in the formal cryptographic sense of indistinguishability. |
| **Recommendation:** The modulo bias should be quantified and shown to be negligible in the security parameter. |

# Re-Examination of System Version 1.3.0

In Sections 3 to 5, we discuss relevant findings in changes made to Swiss Post's e-Voting system in version 1.3.0. Specifically, we examine the changes made to the cryptographic primitives specification document as of version 1.3.0.[3] We focus primarily on the relevant changes highlighted in Swiss Post's Changelog.[4] We additionally review changes made in the system specification document as of version 1.3.0 [5] focusing again on relevant changes highlighted in Swiss Post's CHANGELOG.[6]

**Findings.** As in the previous release (see Section 1.4), the observed changes refine and move the system in a positive direction. In this version, we report on some potential errors in the specification and some more of our recurring themes relating to our recommendation that Swiss Post not only document changes but explain—in writing—its design goals and the necessity and sufficiency of the (new) design.

Our primary concern in this version pertains to Post's proposal that a voter's date of birth be used as an additional authentication factor and incorporated as part of what they refer to as a "shared secret" between the server and voting client (along with the SVK (start voting key) received in the mail).

Although we have not witnessed evidence of a widespread population-level breach of this information in the Canadian context, we have seen some real-world anecdotal examples of dates of birth being compromised for specific individuals [3]. In these instances, a compromised date of birth is henceforth and forever longer *not* secret and, therefore, would offer no meaningful protection on top of the SVK.

At a minimum, the proposal to incorporate date of birth in the voter authentication is essentially a one-off mention in the System specification and requires more detail. Our high-level recommendation, therefore, is to see a broader-ranging conversation and analysis of the real-world security requirements of voter credentials.

[3] Swiss Post Cryptographic Primitives specification version 1.3.0. Available: https://gitlab.com/swisspost-evoting/crypto-primitives/crypto-primitives/-/blob/664b671e35d95af20766cd5f05aef1f58bdbd147/Crypto-Primitives-Specification.pdf

[4] Swiss Post Cryptographic Primitives Changelog for version 1.3.0. Available: https://gitlab.com/swisspost-evoting/crypto-primitives/crypto-primitives/-/blob/664b671e35d95af20766cd5f05aef1f58bdbd147/CHANGELOG.md

[5] Swiss Post Cryptographic System Specification version 1.3.0. Available: https://gitlab.com/swisspost-evoting/e-voting/e-voting-documentation/-/blob/924de3e63a8ebdb962f5acba6964d321a8bc770e/System/System_Specification.pdf

[6] Swiss Post System Specification Changelog for version 1.3.0. Available: https://gitlab.com/swisspost-evoting/e-voting/e-voting-documentation/-/blob/924de3e63a8ebdb962f5acba6964d321a8bc770e/System/CHANGELOG.md

## 4 Changes to Cryptographic Primitives Specification

### 4.1 Clarification to x.509 Certificates

This section primarily concerns changes made to Digital Signatures and x.509 Certificates (Section 6) in regards to Swiss Post's remarks in the Changelog:

> *Change 1A. [Code, Specification] Added additional properties in the x.509 certificates, clarified the description, and improved input validation for digital signatures (feedback from Aleksander Essex).*

**x.509 certificates.** I reviewed the changes against my remarks in Section 2.2 in my 2022 Re-evaluation (Addendum II) from February 13, 2023. Swiss Post has resolved my comments concerning:

- certificate serial numbers
- x.509 version 3
- revocation information
- extensions handling self-signing.
- digital signature algorithm
- hash algorithm

**Digital signatures.** I reviewed Swiss Post's changes relating to my remarks in Section 2.2 in my 2022 Re-evaluation (Addendum II) from February 13, 2023. They resolved my remarks concerning Kleene star notation and arbitrary length outputs of the signature function.

| **Issue 6:** Non-standard hash function in digital signatures (Resolved) |
|---|
| **Description**: As pointed to in my 2023 Re-Examination (Addendum II), Swiss Post continues a non-standard use of `RecursiveHash` in the signature algorithm. |
| **Recommendation:** None (Resolved). Swiss Post specified RSASSA-PSS at the 3072-bit level with SHA-256 as the signature hash function. |

| **Issue 7:** Signature functions should ingest messages, not hashes (Resolved) |
|---|
| **Description**: Currently, Algorithm 6.2 (`GenSignature`) applies the `RecursiveHash` to the message before calling the `Sign()` function. The `Sign()` then applies its own hash function (essentially, a hash of the hash), which seemed redundant. |
| **Recommendation:** None (Resolved). Swiss Post explains in the primitives specification (Section 6.3, version 1.3.0) that the application of `RecursiveHash` prior to the invocation of `Sign` is to avoid having to specify file formats for mixed-format messages. |

| **Issue 8:** Signature scheme should be fully specified |
|---|
| **Description**: In Algorithm 6.2, Swiss Post's specifies the `GenSignature` algorithm outputs a signature of 384 bytes (3072 bits). Where does 384 come from? This seems to necessarily imply `Sign()` is an RSA-based signature. |
| **Recommendation:** If RSA signatures are intended, it should be specified along with the padding scheme used (e.g., PKCS1.5, PSS, etc.) |

| **Issue 9:** Possible typo |
|---|
| **Description**: In Line 4 of Algorithm 6.2, the output of the `Sign()` operation says to "See Algorithm 3.11", i.e., `StringToByteArray()` |
| **Recommendation:** Confirm if this is a typo or intended. |

## 4.2 Clarifications to Argon2id Parameterization

This section primarily concerns changes made to the Argon2id specification (Section 4.5) in regard to Swiss Post's remarks in the Changelog:

> *Change 1B. [Code, Specification] Added a section on Argon2id profiles containing a justification of the chosen parameters (feedback from Rolf Haenni, Reto Koenig, Philipp Locher, and Eric Dubuis).*

**Argon2id Specification.** Swiss Post addressed several of my previous remarks about the Argon2id specification. However, it appears to have introduced some issues.

| **Issue 10:** Typo in Argon2id parameterization profiles |
|---|
| **Description**: Section 4.5 contains a table showing the Argon2id parameterization profiles (`STANDARD, LESS_MEMORY, TEST`). The second column is labeled "Memory (in GiB)," implying that the three profiles use 21, 16, and 14 GiB of memory, respectively. However, this appears to be an error. In Section 4 (Parameter Choice) of RFC9106, Item 1 specifies the standard memory sizes as "m=2^(21) (2 GiB of RAM)." Here, $m$ is the number of kibibytes (i.e., number of units of $2^{10}$ bytes), not to be confused with gibibytes (units of $2^{30}$ bytes).[7] So $2^{21}$ kibibytes $= (2^{21} \cdot 2^{10})/2^{30} = 2^1 = 2$ GiB. |
| **Recommendation:** The column should be changed to: "Memory (in KiB)" and the 21, 16 and 14 should be raised to the power of 2. |

| **Issue 11:** Define term |
|---|
| **Description**: Regarding Argon2id profiles, Swiss Post says: "The first profile STANDARD, ... is considered uniformly safe by RFC9106." It is unclear what "uniformly safe" means in this context, and RFC9106 does not appear to define it either. |
| **Recommendation:** Define "uniformly safe". |

## 5 Changes to System Specification

### 5.1 Voter Authentication

This section primarily concerns changes made to voter authentication made (GetVoter-AuthenticationData, Section 4.1.4) in regard to Swiss Post's remarks in the Changelog:

> *Change 2A. Specified the new voter authentication protocol (feedback from Rolf Haenni, Reto Koenig, Philipp Locher, Eric Dubuis, and Aleksander Essex).*

**Dates of birth for voter authentication.** There are concerns about the suitability of dates of birth in voter authentication. Swiss Post says, "The extended authentication factor `EA_id` like the voter's date of birth can help mitigate the risk of fraudulent voting by making it more difficult for someone to use another person's voting card. Here, the start voting key is being used in combination with the voter's date of birth as a "shared secret." There are several fundamental limitations to this approach:

1. **Low entropy.** Dates of birth are low entropy—about 15 bits in the best case, and lower when factoring in population-level demographics and any partial information known about the target voter. Of course, the entropy is zero bits if the attacker knows the voter's date of birth, which is a plausible scenario given the intersection between the set of a voter's friends and family and those who might reasonably have opportunistic access to a voter's voting card.
2. **Attitudes around secrecy.** Dates of birth are not regarded as a secret by many voters, who share personal information widely across the internet, business services, and social media. It is also not always respected as a secret by governments themselves. For example, the vaccine passport system in various Canadian provinces requires individuals to widely show a QR code containing a digitally-signed attestation of vaccination, which includes the person's date of birth.
3. **Cannot be reissued.** Dates of birth are static and cannot be reissued/reset. They are always one population-level data breach away from completely nullifying their use as a "shared secret" for the entire population for the rest of their lives. At a minimum, this will always apply to some non-empty subset of the people whose dates of birth are already widely known. For those individuals, it will provide no meaningful protection.

| **Issue 12:** Dates of birth as an authentication credential |
|---|
| **Description**: See discussion above. |
| **Recommendation:** As an ideal outcome: find another approach to voter authentication. At a minimum: specify concrete security requirements and provide a cyber-risk assessment of dates of birth as an extended authentication factor. Swiss Post should openly acknowledge these limitations if it intends to pursue this approach. |

I reviewed the the `DeriveCredentialId()` and `DeriveBaseAuthenticationChallenge()`, `GetAuthenticationChallenge()`, and `VerifyAuthenticationChallenge()` functionalities (Section 3.5–Voter Authentication). I also reviewed RFCs 6238 and 4226, which form the basis of the `AuthenticateVoter()` functionality (Section 5.1–Systems specification). Overall, I found no major issues.

| **Issue 13:** Domain separation |
|---|
| **Description**: In Algorithm 3.9 `DeriveBaseAuthenticationChallenge()` Line 2: The extended authentication factor and start voting keys are being concatenated prior to being input to the Argon2id hash. Similar to my comments in Annex 2 (relating to the ElGamal parameter generation), this does not enforce an explicit domain separation.<br><br>For example, an extended authentication factor `EA_id` consisting of a date of birth `1970-01-01` and a 24 character Start Voting Key `ABCDEFG`... would produce the same key $k$ as a year/month of birth string `1970-01-` and a 26 character Start Voting Key `01ABCDEG`... This possibility appears to be precluded by Table 10, which specifies the length `l_SVK` as a fixed 24 `base32` characters. However, once again, as a matter of cryptographic design, a security property like collision resistance should not rely on external requirements such as `l_SVK` and `EA_id` always having the same format. A similar concatenation appears in line 6 of Algorithm 5.1 `GetAuthenticationChallenge`. |
| **Recommendation:** Enforce domain separation of types. |

| **Issue 14:** How is Argon2id less memory profile applied? |
|---|
| **Description**: Several algorithms throughout the systems specification (see e.g., Algorithm 5.1) specify the "less memory" Argon2id profile. In Section 3.6, the following text was added: "In the crypto primitives specification, we define different Argon2id profiles. Each time Argon2id is used, we specify which of these profiles is used." |
| **Recommendation:** Add a brief explanation describing how a memory profile is chosen and what the threat model is. |

## 5.2 Sequence Diagrams

Swiss Post added sequence diagrams in Sections 5.2 and 5.3 with the following remark in the Changelog:

> *Change 2B. Completed the sequence diagrams of the sub-protocols SendVote and ConfirmVote with the messages from the voter authentication protocol (feedback from Rolf Haenni, Reto Koenig, Philipp Locher, Eric Dubuis, and Aleksander Essex).*

I reviewed the sequence diagrams and found no issues or inconsistencies.

### 5.3 Argon2id Profile

Swiss Post made the following remark in the Changelog:

> *Change 2C. Clarified the Argon2id profile used with references to the crypto-primitives-specification (feedback from Rolf Haenni, Reto Koenig, Philipp Locher, Eric Dubuis, and Aleksander Essex).*

See my remarks in Section 4.2.

# Improved Safe Prime Generation

## 6 Design Goals for Safe Prime Generation Algorithms

Sections 6 to 10 discuss potential improvements to Swiss Post's safe prime generation method. The generation of discrete logarithm domain parameters is fundamental not only to the security of election verification but to the perceived legitimacy of the proofs themselves. These parameters must be selected carefully, with minimal degrees of freedom, to reduce the opportunity for cryptographic trapdoors, such as those proposed by Haines et al. [5] in a prior ancestor to the current system.

Swiss Post's high-level approach to verifiable parameter generation has been to use an election's fully qualified name as a seed to a pseudo-random function which outputs complete, valid, and secure parameters for *that* specific election. Overall this approach is sound, although the specific details of which have been a subject of ongoing discussion. I have raised some of these issues consistently throughout the examination process:

1. **Analysis of the Swiss Post e-Voting System**, November 26th, 2021:
    - I recommended specifying a concrete primality testing algorithm for IsProbablePrime()
2. **2022 Re-evaluation of the Swiss Post e-Voting System**, September 30th, 2022:
    - I recommended using Miller-Rabin instead of BPSW as a primality test, as the former has formal guarantees.
3. **2022 Re-evaluation of the Swiss Post e-Voting System (Addendum II)**, February 13th, 2023:
    - I restated my recommendation to use Miller-Rabin for its formal guarantees.

In response to these recommendations, Swiss Post has continued to use the BPSW test but has *added* a Miller-Rabin in system version 1.2.3 (see Section 2). Swiss Post's concern (as they explained to me in direct conversation) relates to performance. In their testing, safe prime parameter generation at the 3072-bit level is slow, and increased runtime is a barrier for them.

To balance these considerations, the purpose of the following sections is to take a more in-depth look into other options for safe prime generation. As a design goal, the objective is a solution offering the following three properties simultaneously:

- Formal guarantees on the probability that a composite number is misidentified as prime,
- Faster runtime than their current proposal,
- Parsimonious design (reliant on external primality testing functionalities only *as necessary*).

## 7 Basic Approaches to Primality testing

### 7.1 Strong Probable Primes and the Miller-Rabin Test

Let $p > 2$ be a positive odd integer and let $a > 1$ be a positive integer such that $a \nmid n$. Let $p - 1$ be written as $2^s d$ such that $d$ is odd. We say $p$ is a *strong probable prime* to base $a$ if either:

$$a^d \equiv 1 \pmod{p},$$

or

$$(a^d)^{2^r} \equiv -1 \pmod{p}$$

for some $0 \leq r < s$. If $p$ is a strong probable prime base $a$ and is composite, then $p$ is called a base-$a$ pseudoprime. The *Miller-Rabin* test to $k$-rounds is the strong probable prime test repeated for $k$ distinct uniform bases $\{a_1, \ldots, a_k\} \leftarrow_{\$} \mathbb{Z}^k$. An integer $p$ passing $k$ such tests is composite with probability less than $(\frac{1}{4})^k$. See e.g., [7] for additional information.

### 7.2 The Baillie-PSW Test

The Baillie-PSW test [2,8] combines two distinct primality tests whose respective pseudoprimes are conjectured to be mostly non-overlapping. An integer $p > 2$ is a *strong BPSW probable prime* if:

- $p$ is a strong probable prime to base 2,
- $p$ is a Lucas probable prime.

A Lucas probable prime[8] is an integer $p$ for which $p$ divides specific elements of a Lucas sequence with an applicable parameterization (such as proposed in [8]). The detailed explanation of the Lucas probable prime test is outside the scope of this document. See Baillie and Wagstaff [2] for further discussion.

Compared to Miller-Rabin, no formal error bounds have been proven for the BPSW test, and infinitely many pseudoprimes are conjectured to exist. To date, however, no counterexamples have been even been found, and the test seems to have gained popularity in recent years due to a perceived performance advantage over Miller-Rabin.

### 7.3 Speed Comparison

When $p$ is composite, the execution time of BPSW is identical to Miller-Rabin for most values of $p$. In that situation, the compositeness of $p$ is witnessed (proven) by a single strong probable prime test, and most composite $p$'s are rejected at that stage.

If $p$ is prime, the relative run times consist of the excess $(k - 1)$ rounds of the strong probable prime testing of Miller-Rabin set against the BPSW's Lucas test. Since BPSW has no formal error bounds, it has no established (proven) equivalence of Miller-Rabin round count $k$. However, as a concrete example, as of version 6.2.0, gmp has begun substituting $k = 24$ Miller-Rabin rounds with the BPSW test.[9] In performance terms, the cost of the Lucas test in BPSW is set against 23 excess rounds of Miller-Rabin.

[8] Not to be confused with Lucas' primality test, which is based on the knowledge of the factorization of $p - 1$.

[9] `https://gmplib.org/gmp6.2`

As a concrete comparison when $p$ is prime $k = 24$, rounds of Miller-Rabin cost approximately 26 times the cost of a single strong probable prime test at the 1000-bit level. By comparison, BPSW costs 4.19 times, making BPSW almost six times faster.[10]

However, because prime generation algorithms typically involve picking (and rejecting) numerous composite candidates before a prime is found, the *aggregate* performance increase is not significant in this setting. For example, at Swiss Post's nominal 3072-bit level, and with trial division of small prime factors up to several tens of thousands, a primality generation algorithm will still generally have to test (and reject) *thousands* of composite candidates before finding a prime. For each rejected candidate, the Miller-Rabin and BPSW tests consist of a single (failed) strong probable prime test and take an identical amount of time.

## 8 Basic Approaches to Generating Safe Primes

In this section, we explore basic approaches to generating safe primes and review the importance of algorithmic efficiency of testing a candidate for small prime factors before the main primality test is applied. A safe prime is a prime of the form $p = 2q + 1$ where $q$ is also prime. Prime $q$ has been historically referred to as a *Sofie Germain* prime.

### 8.1 Naive generation of a $k$-bit safe prime

Let `PsuedoRandom(x,y,z)` be a function that returns an $x$-bit integer according to a pseudo-random function $f_{y,z} : \mathbb{Z} \to \mathbb{Z}$ selected uniformly from the set of $x$-bit pseudo-random functions by the pair $(y, z)$. Let `NoSmallFactors(x, B)` be a function that returns $True$ if input $x$ contains no small prime factors up to a bound $B$. It returns $False$ otherwise. Let `IsPRP(x)` be a probabilistic primality test that returns $True$ if $x$ is a probable prime (according to the particular test implemented inside). It returns $False$ otherwise.

Algorithm 1 presents a basic (naive) algorithm for generating random $k$-bit safe primes. See, e.g., Algorithm 2.86 in Handbook of Applied Cryptography [7].

### 8.2 Swiss Post's Initial Approach to Generating Safe Primes

Swiss Post's initial safe prime generation approach was similar to the naive approach but ran considerably slower as it did not pre-filter candidates containing small factors.

Let `Is_BPSW_PRP` be the Baillie-PSW probable primality test. Swiss Post's initial approach (see primitives specification, version 1.1.0) is depicted in Algorithm 2. Note this is not a verbatim re-statement of the algorithm. It has been simplified in several places for notional and conceptual clarity.

Observe Swiss Post's approach (Algorithm 2) differs from the naive approach Algorithm 1) in three ways. First is that integers are selected deterministically and *pseudo*-randomly, which is not of particular relevance or consequence to this discussion. Second is that Swiss Post concretely implemented their primality test as the Baillie-PSW probabilistic test, which is fast but offers heuristic (instead of formal) bounds on the probabilities.

[10] gmpy2 primality testing benchmarks. Available: https://github.com/aleaxit/gmpy/issues/265

**Algorithm 1:** Naive $k$-bit safe prime generation

```
Input: Bit length $k$, trial division bound $B$, seed value $seed$
Output: $k$-bit safe prime $p$
$i \leftarrow 0$;
while $True$ do
    $i \leftarrow i + 1$;
    $q \leftarrow$ PseudoRandom($k-1, i, seed$) ;   /* Pseudo-random (k-1)-bit int */
    $p \leftarrow 2q + 1$;
    if NoSmallFactors(q, B) then
        if IsPRP(q) then
            if NoSmallFactors(p, B) then
                if IsPRP(p) then
                    return $p$
                end
            end
        end
    end
end
```

**Importance of checking for small factors before primality testing.** Swiss Post's initial approach to safe prime generation was unnecessarily slow due to the absence of a preliminary check of small factors (i.e., `NoSmallFactors`) prior to running the relatively costly BPSW primality test.

Let $P_B = \{3, 5, 7, \ldots, p_n\}$ be the $n$ odd primes less than or equal to some bound $B > 1$. The probability that a randomly chosen (odd) integer $x$ contains no small factors below $B$ is the probability that $x \not\equiv 0 \pmod{p_i}$ for all $p_i \in P_B$, i.e.,

$$P\big(\texttt{NoSmallFactors}(\texttt{x}, \texttt{B}) = True\big) = \prod_{\forall p_i \in P_B} \frac{(p_i - 1)}{p_i}.$$

Using this equation, we can estimate the probability that a random odd integer $x$ contains a small factor below $B$ and hence the probability `NoSmallFactors(x,B)` will reject $x$. Table 1 demonstrates the importance of running `NoSmallFactors()` as a preliminary step for efficiently filtering composite candidates. For example, for $B = 100$, approximately 76 out of 100 candidates can be rejected without running a primality test. The implementation of `NoSmallFactors(x,B)` amounts to running $\gcd(x, \rho_B)$ where $\rho_B$ is the largest odd primorial less than B:

$$\rho_B = \prod_{\forall p_i \in P_B} p_i.$$

Table 2 shows the size of $\rho_B$ in bits for increasing bounds $B$. For example, for $B = 100$, the cost of `NoSmallFactors(x,B)` amounts to a single gcd between $x$ and a 120-bit integer. Optimal bounds for $B$ would depend on $|x|$ weighed against the diminishing returns of running `NoSmallFactors()` on increasing bounds $B$.

**Algorithm 2:** Swiss Post's initial safe prime generation approach (primitives spec. 1.1.0)

```
Input: Bit length $k$, seed value $seed$
Output: $k$-bit safe prime $p$
$i \leftarrow 0$;
while $True$ do
    $i \leftarrow i + 1$;
    $q \leftarrow$ PseudoRandom($k - 1, i, seed$) ;    /* Pseudo-random (k-1)-bit int */
    $q \leftarrow q + 1 - (q \pmod 2)$ ;                        /* Ensure q is odd */
    $p \leftarrow 2q + 1$;
    if Is_BPSW_PRP(q) then
        if Is_BPSW_PRP(p) then
            return $p$
        end
    end
end
```

| $B$ | | | | | | | |
|---|---|---|---|---|---|---|---|
| 25 | 50 | 75 | 100 | 125 | 150 | 175 | 200 |
| 67.3% | 72.3% | 74.8% | 75.9% | 77.0% | 77.9% | 78.6% | 79.2% |

**Table 1.** Probability a random odd integer contains a small factor below bound $B$, i.e., $Pr($`NoSmallFactors(x,B)` $= False)$.

| | $B$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 25 | 50 | 75 | 100 | 125 | 150 | 175 | 200 |
| $\|\rho_B\|$ (bits) | 27 | 59 | 95 | 120 | 154 | 189 | 226 | 272 |

**Table 2.** Computational cost of `NoSmallFactors(x,B)` in terms of $\gcd(x, \rho)$.

## 8.3 Swiss Post's Modified Approach to Safe Prime Generation

Instead of pseudo-randomly generating integers and checking for small factors (via a `NoSmallFactors()` functionality), Swiss Post's modified approach (see primitives spec. 1.2.1) generates an initial pseudo-random candidate and uses a sieving-based approach to *skip over* candidates containing small factors.

Let `Sieve(x,B)` be a function that accepts an integer $x$ and returns the next largest integer that is of the form $6x + 5$[11] and is not a multiple of any number below a bound $B$. Let `MillerRabin(x,k)` return $True$ if $k$ iterations of the Miller-Rabin primality test performed on $x$ show $x$ is a probable prime. Swiss Post's modified approach to safe prime generation is depicted in Algorithm 3.

[11] Sofie Germain primes $q > 7$ are necessarily of the form $6x + 5$. Consequently, safe primes $p$ are of the form $12x + 11$.

**Algorithm 3:** Swiss Post's modified safe prime generation (primitives spec. 1.2.1)

```
Input: Bit length $k$, sieving bound $B$, seed value $seed$
Output: $k$-bit safe prime $p$
$q' \leftarrow$ PseudoRandom($k-1, 0, seed$) ;        /* Pseudorandom (k-1)-bit int */
$q \leftarrow q' - (q' \pmod 6) + 5$ ;                     /* Ensure $q \equiv 5 \pmod 6$ */
while True do
    $q \leftarrow$ Sieve($q, B$) ;       /* Next $q = 6x + 5$ with no divisor below B */
    $p \leftarrow 2q + 1$;
    if Is_BPSW_PRP(q) then
        if Is_BPSW_PRP(p) then
            if MillerRabin(q,64) then
                if MillerRabin(p,64) then
                    return $p$
                end
            end
        end
    end
end
```

## 9 Improved Approaches to Safe Prime Generation

Swiss Post's approach of applying BPSW tests to *both* $p$ and $q$ is mathematically unnecessary. If $q$ is prime, $p$'s primality can be proven in a *single* round of Miller-Rabin, leveraging a recent result from the literature [9], which we show in Theorem 9.3.

We use this observation to propose a novel (to our knowledge) algorithm for safe prime generation in a manner that: (a) offers the formal guarantees of the Miller-Rabin test while simultaneously (b) running slightly faster than a generation algorithm using the Baillie-PSW test (see Algorithm 4). From there, we use Pocklington's theorem to extend this generation algorithm (see Algorithm 6) into one that:

1. Outputs deterministic (as opposed to probabilistic) safe primes.
2. Requires no external primality testing functionalities (Miller-Rabin, Baillie-PSW).
3. Runs slightly faster than all the other approaches presented in this section.

### 9.1 Mathematical Preliminaries

We begin with four theorems relevant to testing whether an integer $p = 2q + 1$ is prime given the primality of $q$. The first theorem establishes a base case for testing in this setting. The second well-known theorem, due to Pocklington, improves on the base case. The third theorem, a recent result due to Ramzy [9], extends Pocklington's theorem to prove that a single Fermat primality test establishes the primality of $p$ given the primality of $q$. Finally, we present a corollary extending Ramzy's result to a single round of the Miller-Rabin test.

**Theorem 9.1 (Primality test of $p$ with known factorization of $p-1$).** *Let $p$ be a integer and let $F = \{f_1, \ldots, f_k\}$ be the set of prime divisors of $(p-1)$. If there exists* *an integer a such that:*

$$\begin{aligned} a^{p-1} &\equiv 1 \pmod{p} \text{ and} \\ a^{(p-1)/f_i} &\not\equiv 1 \pmod{p} \ \forall f_i \in F, \end{aligned}$$

*then p is prime.*

*Proof.* *See Fact 4.38 of Handbook of Applied Cryptography [7].* □

**Remark**: If $q$ and $p = 2q + 1$ are primes, every integer $2 \leq x < (p-1)$ has order either $q$ or $2q$ modulo $p$. For all such integers $x$, clearly $x^{2q} \equiv 1 \pmod{p}$ by Fermat's little theorem and $x^2 \not\equiv 1 \pmod{p}$ since $x \notin \{-1, 1\}$ by definition. Let $a$ be a quadratic non-residue modulo $p$. Then $a$ has order $2q$ and therefore $a^q \not\equiv 1 \pmod{p}$. In other words, if $p$ is a safe prime, half of the values in the range $2 \leq a < (p-1)$ (the set of quadratic non-residues) establish $p$'s primality using this test.

From the perspective of efficiency, however, this approach requires, on average, *two* applications of the test for differing bases $a' \neq a$.

**Theorem 9.2 (Pocklington's theorem for $p$ with large factor of $p-1$).** *Let $p$ be an integer and $q$ be a prime such that $q \mid (p-1)$ and $q > \sqrt{p} - 1$. If there exists an integer $a$ such that*

$$\begin{aligned} a^{p-1} &\equiv 1 \pmod{p} \text{ and,} \\ \gcd(a^{(p-1)/q} - 1, p) &= 1, \end{aligned}$$

*then p is prime.*

*Proof.* *See [6].* □

**Remark**: Applied to the special case of safe primes $p = 2q + 1$, Pocklington's theorem can be further simplified to give us a more efficient test wherein any base in the range $2 \leq a < (p-1)$ can be used to certify the primality of $p$ in a single application of the test. This simplification is given in the following corollary.

**Theorem 9.3 (Ramzy's corollary).** *Let $q$ be prime and let $p = 2q+1$. For any integer $2 \leq a < (p-1)$, if:*

$$a^{p-1} \equiv 1 \pmod{p},$$

*then p is prime.*

*Proof.* If $q$ is prime and $p = 2q + 1$, then the pre-conditions of Pocklington's theorem are satisfied, i.e., $q$ is a prime such that $q \mid (p-1)$ and $q > \sqrt{p} - 1$. If $p$ is prime, clearly the first criterion of Pocklington's theorem (i.e., $a^{p-1} \equiv 1 \pmod{p}$) is satisfied by any integer $a$ as per Fermat's little theorem.

The second criterion, $\gcd(a^{(p-1)/q} - 1, p) = 1$, is equivalent to the statement $a^{(p-1)/q} \not\equiv 1 \pmod{p}$. In the safe prime setting this simplifies to $a^2 \not\equiv 1 \pmod{p}$. This criterion is satisfied by any non-square root of unity in $\mathbb{Z}_p^*$, i.e., for any $2 \leq a < (p-1)$. □

In summary, if $q$ is prime, the primality of $p = 2q + 1$ can be *deterministically* certified in a single Fermat primality test. To our knowledge, this observation was first made only recently (i.e., in the past year) by Ramzy (see Corollary 2.4 of [9]).

This efficient safe prime test can be used to construct a more efficient safe prime generation algorithm. However, since Miller-Rabin is generally more widely implemented than Fermat's test, it is useful to establish that a single round of Miller-Rabin could take the place of the Fermat test in a practical implementation.

**Theorem 9.4 (A single-round Miller-Rabin safe prime test).** *Let $q$ be a prime and let $p = 2q + 1$. For any integer $2 \leq a < (p - 1)$, if $p$ is a base-$a$ strong probable prime, $p$ is prime.*

*Proof.* By Ramzy's corollary, if $p$ is a base-$a$ Fermat probable prime, $p$ is prime. The proof proceeds by showing if $p$ is a base-$a$ strong probable prime, $p$ is a base-$a$ Fermat probable prime.

Recall an integer $p$ is a strong probable prime base $a$ if:

$$a^q \equiv 1 \pmod{p}$$

or there exists some $0 \leq r < s$ such that

$$a^{2^r q} \equiv -1 \pmod{p}.$$

In the safe prime setting, $s = 1$. Therefore the second criterion reduces to a single case:

$$a^q \equiv -1 \pmod{p}.$$

In other words, $p$ is a base-$a$ strong probable prime if:

$$a^q \equiv b \pmod{p}$$

for $b \in \{-1, 1\}$. Squaring both sides, we have:

$$\begin{aligned}(a^q)^2 &\equiv b^2 \pmod{p}\\ a^{2q} &\equiv 1 \pmod{p}\\ a^{p-1} &\equiv 1 \pmod{p}.\end{aligned}$$

Therefore $p$ is also a base-$a$ Fermat probable prime. Therefore $p$ is prime (specifically a safe prime) by Ramzy's corollary. □

## 9.2 A Faster Primality Test of Safe Prime $p$

We propose a new algorithm for safe prime generation leveraging Ramzy's corollary to replace the full primality test of safe prime candidate $p$. Our approach is to test $p$ first: We apply a strong probable prime test (i.e., single round Miller-Rabin) on $p$ to a fixed base (without loss of generality, we use 2). See Algorithm 4.

This approach provides two benefits relative to Swiss Post's improved approach (Algorithm 3). Firstly, there is now no longer a performance benefit to using BPSW; any candidate $p$ passing the initial strong probable prime test does not require the additional $(k - 1)$ rounds of the naive approach if $q$ is subsequently found to be prime.

**Algorithm 4:** Safe prime generation with faster primality test of $p$

```
Input: Bit length $k$, trial division bound $B$, seed value $seed$, minimum required
       rounds of Miller Rabin $\lambda$
Output: $k$-bit safe prime $p$
$i \leftarrow 0$;
while $True$ do
    $i \leftarrow i + 1$;
    $p' \leftarrow$ PseudoRandom($k, i, seed$) ;        /* Pseudorandom k-bit integer */
    $p \leftarrow p' - (p' \pmod{12}) + 11$ ;           /* Ensure $p \equiv 11 \pmod{12}$ */
    $q \leftarrow p \gg 1$ ;    /* Logical right shift computing $q \leftarrow (p-1)/2$ */
    if NoSmallFactors(p,B) then
        if NoSmallFactors(q,B) then
            if MillerRabin(p,1) then
                if MillerRabin(q,$\lambda$) then
                    return $p$
                end
            end
        end
    end
end
```

**Theorem 9.5 (Correctness of Algorithm 4).** *Let $p = 2q + 1$ be the output of Algorithm 4.*

$$Pr(p \text{ is a safe prime}) > 1 - \left(\frac{1}{4}\right)^{\lambda}.$$

*Proof.* By definition, the probability that $p$ is a safe prime is equivalent to the joint probability that $p$ and $q$ are prime:

$$P(p \text{ is prime } \cap\ q \text{ is prime}) = P(p \text{ is prime} \,|\, q \text{ is prime}) P(q \text{ is prime})$$

Algorithm 4 returns $p$ (resp. $q$) if and only if $\texttt{MillerRabin}(\texttt{p}, \texttt{1}) = True$ and $\texttt{MillerRabin}(\texttt{q}, \lambda) = True$. Given this,

$$Pr(p \text{ is prime} \,|\, \text{q is prime}) = 1$$

as per Theorem 9.4, and

$$Pr(q \text{ is prime}) > 1 - \left(\frac{1}{4}\right)^{\lambda}$$

as per standard error-bounds of Miller-Rabin (see, e.g., Fact 4.25 of [7]). □

**Performance of Algorithm 4.** In Swiss Post's improved algorithm (Algorithm 3), each candidate $q$ (and subsequently $p$) is subjected to a BPSW test. Additionally, as a final check, $k = 64$ rounds of Miller-Rabin are applied. By comparison, if Algorithm 4 finds $p$ to be a strong probable base-2 prime, we need not apply the Lucas test (as in BPSW). Rather, we can immediately move on to testing $q$. For this, we use Miller-Rabin *only*.

For most composite values of $p$, both Algorithms 3 and 4 will reject $p$ at the initial base-2 strong probable prime test and therefore will have the same runtime in such cases.

Relative to Swiss Post's approach, Algorithm 4 completes primality testing on $p$ faster. When $p$ is found to be a base-2 strong probable prime, Algorithm 4 moves on to testing $q$ with the Miller-Rabin test, while Algorithm 3 continues into the Lucas test and eventually moves on to the $BPSW$ test on $q$. If $q$ is prime, the BPSW test is faster. But this speed-up is only realized on the final (winning) candidate out of the thousands of candidates expected at the 3072-bit level. Based on our benchmarking experiments, Algorithm 4 is slightly faster than the Swiss Post approach—by approximately 10%.

### 9.3 Fast Deterministic Generation Based on Pocklington's Theorem

Recall Pocklington's theorem allows us to efficiently and *deterministically* test the primality of a number $n$ when a large prime factor of $n-1$ is known. In this section, we use Pocklington's theorem (see Theorem 9.2) to improve the performance of Algorithm 4. We proceed in two steps. The first step generates $q$ based on what will become a large prime factor $s|(q-1)$. The second step extends this approach toward its conclusion: beginning with a known prime and incrementally and deterministically generating larger primes from it.

**Faster Pocklington-based generation of $q$.** The high-level approach has three components:

1. Generate a prime $s$ of size $\lceil \frac{|p|-1}{2} \rceil$ by calling a subroutine `PseudoRandomPrime(k,seed,` $\lambda$`)` which outputs a pseudo random $k$-bit prime using a standard primality generation approach, such as Algorithm 1 with $\lambda$ rounds of Miller-Rabin.
2. Iteratively generate random (even) integers $r$ of size $\lfloor \frac{|p|-1}{2} \rfloor$ and apply Pocklington's test on $q = rs + 1$ to test the primality of $q$.
3. Test $p = 2q + 1$ for primality using a strong probable primality test (as per Theorem 9.4).

This approach is shown in Algorithm 5. Our tests show this approach runs slightly faster than Swiss Post's modified approach (Algorithm 3). In the Swiss Post version, the candidate $q$ is generated randomly, and the initial round of the Miller-Rabin strong probable prime test requires a full modular exponentiation of length $|q|$ when $q \equiv 3 \pmod 4$ (i.e., when $(q-1)/2)$ is odd). By contrast, we can check the second element of Pocklington's criterion (i.e., if $\gcd(2^s \pmod q - 1, q) = 1$), which requires a shorter modular exponentiation of length $|q|/2$. Approximately half of the candidates of $q$ can be eliminated at this stage without proceeding to the full $|q|$ exponentiation resulting in an overall speed-up of approximately 25%.

**Deterministic generation.** In this section, we extend the generation of $q$ to its natural conclusion by using Pocklington's theorem, to begin with a small known prime and iteratively generate successively larger primes up to $|q|$. The result is a safe prime generation algorithm that is fully deterministic, i.e., the probability that $q$ or $p$ is composite is *zero*. A related approach for DSA groups is described in FIPS 186-5 Appendix B.10 [1]. Our high-level approach is as follows:

**Algorithm 5:** Faster Pocklington-based generation of $q$

```
Input: Bit length $k$, trial division bound $B$, seed value $seed$, minimum required
       rounds of Miller Rabin $\lambda$
Output: $k$-bit safe prime $p$
$i \leftarrow 0$;
$s \leftarrow$ PseudoRandomPrime($\lceil k/2 \rceil, seed, \lambda$) ;                /* Generate prime */
while $True$ do
    $i \leftarrow i + 1$;
    $r' \leftarrow$ PseudoRandom($\lfloor k/2 \rfloor, i, seed$) ;           /* Pseudorandom integer */
    $r \leftarrow r - (r \pmod 2)$ ;                                      /* $r$ is even */
    $q \leftarrow rs + 1$;
    $p \leftarrow 2q + 1$;
    if $q \pmod 6 = 5$ then
        if NoSmallFactors(r,B) then
            if NoSmallFactors(q,B) then
                if NoSmallFactors(p,B) then
                    $x \leftarrow 2^r \pmod q$ ;                           /* Pocklington test */
                    if $\gcd(x-1, q) = 1$ then
                        if $x^s \pmod q = 1$ then
                            if $2^{p-1} \pmod p = 1$ then
                                return $p$ ;                       /* Ramzy's Corollary */
                            end
                        end
                    end
                end
            end
        end
    end
end
```

1. Begin with a small known prime $s$.[12]
2. Select random even integers $b < t < s$, until Pocklington's criterion finds $u = st + 1$ is prime.
3. If $|u| = |q|$, output $u$. Otherwise, set $s \leftarrow u$ and return to Step 2, adjusting the lower bound $b$ appropriately in the final iteration to ensure $|u| = |q|$.
4. Run Algorithm 5 with $s \leftarrow u$ instead of $s \leftarrow$ `PseudoRandomPrime()`.

This approach is shown in Algorithm 6.

## 10 Comparison of Safe Prime Generation Techniques

As a benchmark comparison, we implemented each safe prime generation algorithm. Tests were conducted on a 3.6GHz 8-Core Intel Core i9 using Python 3.8, `gmpy2` 2.1.5, and `gmp` 6.2.1.[13]

[12] The primality of $s$ can be efficiently proven, e.g., through exhaustive trial division.

[13] `gmpy2` Python extension for gmp. Available: https://pypi.org/project/gmpy2/
`gmp` multi-precision arithmetic library. Available: https://gmplib.org/

**Algorithm 6:** Fully deterministic generation of $p, q$

```
Input: Bit length $k$, trial division bound $B$, seed value $seed$
Output: $k$-bit safe prime $p$
1  $i \leftarrow 0$;
2  $seed\_primes = [\dots]$ ; /* List of small, efficiently provable primes */
3  $s \leftarrow$ PseudoRandomChoice($seed\_primes, seed$)
4  while True do
5      $i = i + 1$;
6      $b \leftarrow \lfloor s/2 \rfloor$ ;            /* Adjust so $|u| = |k-1|$ in the final round */
7      $t' \leftarrow$ PseudoRandom($b, s, i, seed$) ;  /* Pseudorandom integer $b \leq t' < s$ */
8      $t \leftarrow t' - (t \pmod 2)$ ;                                   /* $t$ is even */
9      $u \leftarrow ts + 1$;
10     if NoSmallFactors(u,B) then
11         $x \leftarrow 2^t \pmod u$ ;                               /* Pocklington test */
12         if $\gcd(x-1, u) = 1$ then
13             if $x^s \pmod u = 1$ then
14                 if $|u| = (k-1)$ then
15                     break ;                        /* $u$ is prime and $|u| = k-1$ */
16                 else
17                     $s \leftarrow u$
18                 end
19             end
20         end
21     end
22 end
23 $p \leftarrow$ Algorithm5($k, B, seed$) ; /* Run Algorithm 5 with the $u$ found above,
     replacing Line 1 with $s \leftarrow u$. */
24 return $p$;
```

For each algorithm, we generated 100 safe primes at the $|p| = 3072$ bit level and recorded the average (mean) runtime. For the trial division step, we computed the greatest common divisor between the prime candidate and a $B$-primorial (product of distinct primes up to bound $B$). We chose $B = 20k$ where $k$ is the bit length of the given prime being generated (which varies in the case of Algorithms 5 and 6). The approach was heuristically chosen through experimentation to find the optimal runtime on the test platform.

As an additional implementation note, the list of small provable `seed_primes` in Algorithm 6 must not be *too* small. Otherwise, there may be insufficiently many options for $t$ in the first iteration of the loop to yield a prime $u = ts + 1$.

### 10.1 Results and Findings

Our results are shown in Table 3. Our findings suggest that Pocklington-based deterministic prime generation is a promising approach to satisfying our design goals set out in Section 6. In particular, Algorithm 6 offers: (1) formal, deterministic guarantees, (2) faster run times (29% faster than Swiss Post's modified approach)[14] and, (3) a simpler, self-contained design with no reliance on external primality testing software libraries.

| Safe Prime Generation Algorithm | Mean runtime (s) | External Testing Dependencies | Formal Guarantees |
|---|---|---|---|
| Swiss Post Initial (Algorithm 2) | Not tested | Baillie-PSW | None ($p$ and $q$) |
| Swiss Post Modified (Algorithm 3) | 182 (100%) | Baillie-PSW and Miller-Rabin | Probabilistic error bounds in $p$ and $q$ |
| Faster primality test on $p$ (Algorithm 4) | 164 (90%) | Miller-Rabin | Probabilistic error bounds in $q$ |
| Faster Pocklington-based generation of $q$ (Algorithm 5) | 135 (74%) | Miller-Rabin | Probabilistic error bounds in large factor of $(q-1)$ |
| Deterministic generation of $p$ and $q$ (Algorithm 6) | 129 (71%) | None | Deterministic |

**Table 3.** Comparison of speed, implementation complexity, and formal security guarantees of various safe prime generation strategies. Benchmarks are an average of 100 tests at the $|p| = 3072$ bit level.

## References

1. Digital signature standard (dss). *FIPS 186-5*, 2023.
2. R. Baillie and S. S. Wagstaff. Lucas pseudoprimes. *Mathematics of Computation*, 35:1391–1417, 1980.
3. A. Cardillo, N. Akinyokun, and A. Essex. Online voting in ontario municipal elections: A conflict of legal principles and technology? In *International Joint Conference on Electronic Voting*, pages 67–82. Springer, 2019.
4. L. Chen, D. Moody, K. Randall, A. Regenscheid, and A. Robinson. Recommendations for discrete logarithm-based cryptography: Elliptic curve domain parameters. *NIST Special Publication 800-186*, 2023.
5. T. Haines, S. J. Lewis, O. Pereira, and V. Teague. How not to prove your election outcome. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 644–660, 2020.
6. N. Koblitz. *A Course in Number Theory and Cryptography*, volume 144 of *Graduate Texts in Mathematics*. Springer, 1994.
7. A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography*. CRC press, 1996.
8. C. Pomerance, J. L. Selfridge, and S. S. Wagstaff. The pseudoprimes to $25 \cdot 10^9$. *Mathematics of Computation*, 35:1003–1026, 1980.
9. A. Ramzy. A primality test for $Kp^n + 1$ numbers and a generalization of Safe primes and Sophie Germain primes. *Notes on Number Theory and Discrete Mathematics*, 29(1):62–77, 2023.

[14] Faster is defined here as the percentage decrease in average runtime.

## A Implementation of Swiss Post's Modified Generation (Algorithm 3)

Python implementation of Swiss Post's improved safe prime generation algorithm. Note: For a more direct comparison, we did not implement Swiss Post's sieving approach.

```python
import gmpy2
import random

B = k * 20  # Trial division bound -- heuristically chosen
primorial = gmpy2.primorial(B) // 2  # Odd B-primorial
random.seed('Election Name Goes Here')


def generate_safeprime(k=3072):
    while True:
        q_prime = gmpy2.mpz(random.randrange(2**(k-1),  2**k)) >> 1
        q = q_prime - (q_prime % 6) + 5
        p = 2 * q + 1

        # Trial division (instead of sieving)
        if gmpy2.gcd(q, primorial) == 1:
            if gmpy2.gcd(p, primorial) == 1:
                # Baillie-PSW tests
                if gmpy2.is_bpsw_prp(q):
                    if gmpy2.is_bpsw_prp(p):
                        # Additional Miller-Rabin tests
                        if gmpy2.is_prime(q, 64):
                            if gmpy2.is_prime(p, 64):
                                    return p
```

## B Implementation of Faster Primality Test of $p$ (Algorithm 4)

Python implementation of safe prime generation with deterministic primality testing of $p$.

```python
import gmpy2
import random

B = k * 20  # Trial division bound -- heuristically chosen
primorial = gmpy2.primorial(B) // 2  # Odd B-primorial
random.seed('Election Name Goes Here')


def generate_safeprime(k=3072):
    while True:
        pp = gmpy2.mpz(random.randrange(2**(k-1),  2**k))
        p = pp - (pp % 12) + 11
        q = p >> 1

        # Trial division
        if gmpy2.gcd(p, primorial) == 1:
            if gmpy2.gcd(q, primorial) == 1:
                # Ramzy's corollary test on p
                if gmpy2.is_strong_prp(p, 2):
                    # 64 rounds of Miller-Rabin on q
                    if gmpy2.is_prime(q, 64):
                            return p
```

## C Implementation of Partial Pocklington Test on $q$ (Algorithm 5)

Python implementation of safe prime generation using deterministic primality testing of $p$ and probabilistic generation of $q$ using Pocklington's theorem.

```python
import gmpy2
import random

B = k * 20  # Trial division bound -- heuristically chosen
primorial = gmpy2.primorial(B)//2  # Odd B-primorial
random.seed('Election Name Goes Here')

def generate_s(k, s_min, s_max):
    while True:
        s = gmpy2.mpz(random.randrange(s_min, s_max))
        s = gmpy2.bit_set(rp, 0)  # Odd s
        # Trial division
        if gmpy2.gcd(s, primorial) == 1:
            #  64 rounds of Miller-Rabin on s
            if gmpy2.is_prime(s, 64):
                return s


def generate_safeprime(k=3072):
    s_max = 2 ** (k//2)
    s_min = s_max // 2

    s = generate_s(k, s_min, s_max)  # Generate seed prime s
    r_max = 2**(k-1) // s    # r < s per Pocklington's criterion
    r_min = 2**(k-2) // s

    while True:
        rp = gmpy2.mpz(random.randrange(r_min, r_max))
        r = gmpy2.bit_clear(rp, 0)  # Even r
        q = r * s + 1
        p = 2 * q + 1

        if (q % 6) == 5:
            # Trial division
            if gmpy2.gcd(q, primorial) == 1:
                if gmpy2.gcd(p, primorial) == 1:
                    # Pocklington Primality test of q
                    x = pow(2, r, q)
                    if gmpy2.gcd(x-1, q) == 1:
                        if pow(x, s, q) == 1:
                            # Ramzy's corollary on p
                            if pow(2, p-1, p) == 1:
                                return p
```

## D Implementation of Fully Deterministic Generation (Algorithm 6)

Python implementation of safe prime generation using fully deterministic generation of $q$ (Pocklington's theorem) and deterministic primality testing of $p$ (Ramzy's corollary).

```python
import gmpy2
import random

random.seed('Election Name Goes Here')
seed_primes = [101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167,
↪ 173, 179, 181, 191, 193, 197, 199]


def pocklington_prime(s, s_min, s_max):
    if s ** 2 < s_min:
        t_max = s
        t_min = s // 2
    else:
        t_max = s_max // s
        t_min = s_min // s

    B = 2 * gmpy2.bit_length(s) * 20  # Output has length |u| = 2*s
    primorial = gmpy2.primorial(B)//2

    while True:
        tp = gmpy2.mpz(random.randrange(t_min,  t_max))
        t = gmpy2.bit_clear(rp, 0)  # Even t
        u = t * s + 1

        # Trial division
        if gmpy2.gcd(u, primorial) == 1:
            x = pow(2, t, u)
            # Pocklington primality test of u
            if gmpy2.gcd(x-1, u) == 1:
                if pow(x, s, u) == 1:
                    return u


def generate_s(k, s_min, s_max):
    s = random.choice(seed_primes)

    while True:
        s = pocklington_prime(s, s_min, s_max)
        if s > s_min:
            return s


def generate_safeprime(k=3072):
    s_max = 2 ** (k//2)
    s_min = s_max // 2

    s = generate_s(k, s_min, s_max)  # Deterministically generate s

    r_max = 2**(k-1) // s  # r < s per Pocklington's criterion
    r_min = 2**(k-2) // s


    while True:
        rp = gmpy2.mpz(random.randrange(r_min, r_max))
        r = gmpy2.bit_clear(rp, 0)  # Even r
        q = r * s + 1
        p = 2 * q + 1

        if (q % 6) == 5:
            # Trial division
            if gmpy2.gcd(q, primorial) == 1:
                if gmpy2.gcd(p, primorial) == 1:
                    # Pocklington Primality test of q
                    x = pow(2, r, q)
                    if gmpy2.gcd(x-1, q) == 1:
                        if pow(x, s, q) == 1:
                            # Ramzy's corollary on p
                            if pow(2, p-1, p) == 1:
                                return p
```