Final Year Project Report

MSci - Interim Report

Implementing PKCS#1 v1.5 and other deterministic RSA Signature Schemes with provably secure parameters

Jude Asare

A report submitted in part fulfilment of the degree of

MSci (Hons) in Computer Science (Information Security)

Supervisor: Saqib Kakvi

Department of Computer Science
Royal Holloway, University of London

April 4, 2024

Table of Contents

Chapter 1: **Introduction**

1.1 Aims

1.1.1 Why: The Context and Rationale

The PKCS#1 v1.5 digital signature scheme, rooted in RSA's hash-and-sign framework, has emerged as the de facto standard for digital signatures since its 1998 introduction [1]. Essential in security-focused network protocols like SSH, DNSSEC, IKE, and pre-TLS 1.3 X.509 certificates [2], its simplicity fostered broad adoption from its outset with a straightforward implementation across programming languages, and speedy verifications relative to alternatives like DSA or ECDS [3]. Despite its widespread adoption the scheme lacks formal security proofs, raising concerns about its long-term reliability. Given its deep integration in various applications, even at the hardware level, transitioning to provably secure [4], [5] alternatives like RSA-PSS has been slow (RSA-PSS was only upgraded to a requirement for new applications in PKCS#1 v2.2 [6] long after initially being suggested as the replacement for PKCS#1 v1.5 in PKCS#1 v2.1 [7]) with imperative requirements of backward compatibility and interoperability acting as significant barriers to full adoption.

Additionally considered in a broader sense, for a cryptographic scheme i.e., RSA-PSS to gain traction it must first be developed and then subjected to rigorous scrutiny by the cryptographic community. Following this, its algorithmic primitives need to be standardised, which then facilitates the creation and standardisation of high-level protocols incorporating these components. Subsequent software implementations must align with these standards before the scheme can be set as the default in various applications. RSA-PSS, while superior to the older PKCS#1 v1.5 lingers in the early stages of this transition mainly between standardisation of its algorithm primitives and the standardisation of protocols that incorporate its primitives. The current landscape is one of fragmentation where RSA-PSS and PKCS#1-v1.5 often coexist within the same infrastructure, such as TLS implementations and RSA certificates. This state of the art considered, retaining PKCS#1 v1.5 scheme remains a preferable choice. Going even further, finding an applicable security proof to bridge its security to the level of its widespread adoption would be the ideal scenario.

In this direction a landmark development came in 2018 when Jager, Kakvi, and May [3] provided a security proof (an improvement on Coron's proof that was limited to the Rabin-Williams variant with e=2 [8]) for the PKCS#1 v1.5 signature scheme, with a caveat: usage of larger parameters are required. Their work offered insights that also apply to other deterministic (and standardised) RSA signature schemes, including ISO/IEC 9796-2 Scheme 1 and ANSI X9.31 rDSA.

1.1.2 What: The Primary Goal

The aim of this project was to connect the theoretical concepts introduced by Jager et al. [3] with practical application. This involved implementing provably secure parameters and then assessing the burden on computational time that arises when using them in deterministic RSA signature schemes.

1.1.3 How: Overview of Objectives

The project pursued a methodical approach to assess the impact on computational time of employing provably secure parameters in deterministic RSA signature schemes. This was achieved by developing algorithms respective to both cases (standard vs. provably secure parameters) and comparing them for various key sizes when applied to large batches of data (in the case of signature operations) or in the case of key generation, generating keys repeatedly for each parameter type at scale. The investigation spanned across various signature scheme standards as means of enhancing the reliability and accuracy of the findings by offering views into the computational overhead from within the context of different schemes. The final deliverable constituted an all-incorporative user-interfaced benchmarking program for which results from each of the signature schemes working on batches of data for multiple key sizes were generated.

1.1.4 Overview of Results

Overall, the results indicated that the integration of provably secure parameters into deterministic signature schemes, far from being a computational burden, can rather lead to substantial improvements in verification times (hash function dependent), while there is no observed impact on signature creation and the impact on key generation remains minimal becoming negligible for larger key sizes.

1.2 Objectives

To achieve the aim, the project involved:

- Exploring the security proof relevant to the PKCS#1 v1.5 signature scheme .
- Determining the practical implications arising from the security proof on instantiations of the full suite of deterministic signature schemes (PKCS#1-v1.5, ISO/IEC-9796-2, and ANSI-X9.31 signature schemes).
- Developing concrete implementations of the considered deterministic signature schemes using standard and provable parameters.
- Demonstrating in practice, with larger parameters, how deterministic schemes can achieve a security level on par with less-practical, signature schemes, such as RSA-PSS.
- Evaluating and comparing computational time of instantiating the considered signature schemes with standard vs. provable parameters.
- Creating a user-interfaced benchmarking program that incorporates all objectives of the project to assess the overhead across various key sizes for the different deterministic signature schemes.

1.3 Organisation

The structure of this paper is outlined as follows: Chapter 2 presents related work on the provable security of deterministic schemes. Chapter 3 covers the background theory for RSA signature schemes. Chapter 4 discusses the proof theorems and draws out their practical implications. Chapter 5 explores assumptions used in implementation, these being derived from the implications presented in Chapter 4. Chapter 6 details the complete development of the project's deliverable. Chapter 7 presents the results, coupled with summaries of key findings. Finally, Chapter 8 concludes the paper with a synthesis of the discussed points.

Chapter 2: **Literature Review/Related Work**

RSASSA-PKCS1-v1.5 remains unbroken. There are no real attacks able to successfully exploit the scheme free of implementation errors. This distinction of being free of implementation errors is crucial. Potential proofs consider only forgeries that are accepted by a correct verification algorithm. It is now well established from a variety of follow up studies all originating from [9] that vulnerable implementations of a flawed signature verification algorithm for RSASSA-PKCS1-v1.5 can be exploited. Bleichenbacher presented a low-exponent attack on RSA-PKCS#1 v1.5 signatures at the CRYPTO 2006 rump session. This attack was later described by Finney [10] in a posting to the OpenPGP mailing list.

It was not until the efforts of Coron in 2002 ([8]) that a security proof applicable to RSASSA-PKCS1-v1.5 arrived. This was due to the issue of deterministic padding scheme that RSASSA-PKCS1-v1.5 uses, rendering standard proof techniques void. Coron presented a security proof for RSASSA-PKCS1-v1.5 (and ISO/IEC 9796-2 signatures) albeit with a restriction that e = 2, i.e. the Rabin-Williams variant [8] which is secure based on the factoring assumption. The proofs' exclusive and/or restrictive value of e aside, a further caveat was that the output size of the hash function needed to be 2/3 the bit length of the modulus N. These restrictions diverge largely from the standard parameters that could potentially be used in the instantiation of RSA-PKCS#1 v1.5 signatures in practice.

Much later, Jager, Kakvi and May [3] showed an improved security proof for RSASSA-PKCS1-v1.5 with less restrictive conditions. It sufficed that e more generally be a small prime (Kakvi and Kiltz [11]). Still requiring a large hash function output, the authors achieved an improvement in hash function output, requiring only 1/2 of the modulus size. The modulus effectively doubles in bit length when compared to the norm with a newly introduced third prime factor necessitating the increase in bits. Withstanding the improvement and as a consequence of the proofs being founded in the random oracle model, the larger cryptographic parameters still deviate slightly from the standard parameters that could be used in practice. Nonetheless this was sufficient enough for the authors to demonstrate how RSA-PKCS#1 v1.5 signatures can be instantiated in practice such that the improved proofs apply.

To all intents and purposes, regardless of the proofs being presented primarily for RSASSA-PKCS1-v1.5, uniformity in construction philosophy means other signature standards of the same deterministic RSA type (ISO/IEC 9796-2 signatures and ANSI X9.31 rDSA) still match the setting required for proofs to be applicable to them. For example theorem statements used in construction of Corons' proof [8] are general enough that corresponding proof theorems can also be presented for the remaining standards of deterministic signatures.

A full discussion of provable security extending to non-deterministic schemes lies beyond the scope of this project. Probabilistic padding poses an issue since generating sources of randomness, particularly on constrained devices is an ongoing challenge. Moreover RSA-PSS, the sole RSA-based randomised digital signature scheme uses two hash functions. This makes it difficult to compare with deterministic schemes that use only one. This project focused on standardised deterministic schemes which are directly comparable and subversion resistant [12] by default of not having to generate randomness. This lends well to the issue of examining computational overhead from various perspectives of the different deterministic standards.

Chapter 3: **Cryptographic Foundations**

3.1 Notations

The security parameter is denoted as λ. This parameter determines a system's security level or key sizes. A larger value of λ indicates stronger security, though it requires more computational effort. The concept of binary strings is frequently referred to. Specifically for all $n \in \mathbb{N}$, the symbol 1^n represents the n-bit string of consecutive ones.

In cryptographic operations, sampling often involves randomness. Given any set S, the notation $x \in_R S$ signifies that x is chosen uniformly at random from S. The set of prime numbers is represented as $\mathbb{P}$ and the set of k-bit primes is denoted as $\mathbb{P}[k]$. Similarly, the set of integers is represented by $\mathbb{Z}$ and $\mathbb{Z}[k]$ respectively.

The notation $\mathbb{Z}_N^*$ represents the multiplicative group modulo N where $N \in \mathbb{N}$.

Finally, game-based proofs are employed, and the notation $G^A \Rightarrow 1$ indicates an event where the adversary A succeeds in game G, specifically when the Finalise Procedure yields an output of 1.

3.2 Digital Signature Schemes

In the realm of security, not every issue revolves around confidentiality. In many instances, the adversaries are not limited to passive surveillance but can be active opponents capable of manipulating or introducing unauthorised messages into a communication stream. In the public key setting, the cryptographic primitive used to provide not only data integrity but also for ensuring the authenticity and non-repudiation of communications is a digital signature scheme. Essentially, Digital signatures function as a means of binding an identity to specific information. A signature process consists of transforming the relevant message and the entity's confidential details to produce tag named a signature. This process serves as a verifiable stamp of authenticity, confirming that the message genuinely originated from the claimed source. Moreover it addresses the concept of non-repudiation. Once a message is signed, the signer is bound to the act of signing and cannot deny having signed a then sent message.

An everyday application of digital signatures is in the domain of web security, specifically in the use of Public Key Infrastructure (PKI) and Certificate Authorities (CAs) for verifying the authenticity of websites. For example, when a user visits a website, the site's digital certificate, issued by a CA, is used to establish a secure connection. The digital signature on the certificate verifies that the certificate is indeed issued by a legitimate Certificate Authority and has not been tampered with. This ensures that the website the user is connecting to is actually the one it claims to be, and not a fraudulent site attempting to impersonate it.

Definition 3.1. A (digital) signature scheme DS consists of three probabilistic polynomial-time algorithms (Gen, Sign, Verify) defined as follows:

1. Gen (key-generation algorithm): takes as input the security parameter 1^λ and outputs a pair of keys (sk, vk), where sk is the signing key and vk is the verification key.

2. Sign (signing algorithm): takes as input a private key sk, message m and outputs a signature σ ($\sigma \leftarrow \text{Sign}_{sk}(m)$).

3. Verify (deterministic verification algorithm): takes as input a public key pk, message m, signature σ and outputs a boolean ($b := \text{Vrfy}_{pk}(m, \sigma)$).

The correctness of DS can be confirmed if for any $\lambda \in \mathbb{N}$ and all (pk, sk) output by Gen(1^λ), it holds that:

$$Pr[\text{Verify(pk, m, Sign}(sk, m)) = 1]$$

Such a signature scheme can be utilised in the following way. Bob, the sender, initiates the process by running Gen(1^λ), which generates a pair of keys, specifically the private key (sk) and the public key (vk). He then discloses his public key (vk) to all, including Alice, the intended receiver.

When the need arises for Bob to send a message m to Alice, he creates a signature σ using his private key: $\sigma \leftarrow \text{Sign}_{sk}(m)$. This signature, along with the message itself, (m, σ), is then dispatched to Alice.

Upon receiving the message and its corresponding signature (m, σ), Alice, already in possession of Bob's public key, proceeds to verify the authenticity of the message. This is done by checking whether $\text{Vrfy}_{vk}(m, \sigma) \stackrel{?}{=} 1$. This process assures Alice of the discussed properties on the message and/or its initiator Bob.

3.3 Security of signature schemes

Security for a signature scheme, represented as DS = (Gen, Sign, Vrfy), is depicted through a contest between a challenger and an Adversary A (probabilistic machine operating within polynomial time). This contest emulates a situation in which A endeavours, to compromise the signature scheme by employing a specific attack model.

3.3.1 Default notion of Secure Signatures

The intuitive idea behind the default notion of security for digital signature schemes is that no efficient adversary should be able to generate valid digital signature for any "new" document that was not previously signed by the original signer. However, adversaries may gain access to all documents and their corresponding signatures through a sign oracle, and they may also exert influence over the content of documents, thus enabling replay attacks.

A robust digital signature system, resistant to such forgeries, is termed "existentially unforgeable under an adaptive chosen-message attack". "Existentially unforgeable" signifies that the adversary cannot produce a valid signature on any document. The protection remains intact even if the adversary can carry out an adaptive chosen-message attack (hence the latter phrasing) by which it is able to obtain signatures on arbitrary messages chosen adaptively during its attack.

Game UF-CMA(ROM)

Initialise

$(pk, sk) \xleftarrow{\$} \text{Gen}(1^\lambda)$

return pk

Hash(m)

if $(m, \cdot) \in \mathcal{H}$

  fetch $(m, y) \in \mathcal{H}$

  return y

else

  $y \in_R$ Domain;

  $\mathcal{H} \leftarrow \mathcal{H} \cup \{(m, y)\}$

  return y

Sign(m)

$\mathcal{M} \leftarrow \mathcal{M} \cup \{m\}$

return $\sigma \xleftarrow{\$} \text{Sign}(sk, m)$

Finalise(m^*, σ^*)

if $\text{Vrfy}(pk, m^*, \sigma^*) == 1 \wedge m^* \notin \mathcal{M}$

  return 1

else

  return 0

Figure 3.1: Game defining UF-CMA security in the Random Oracle Model (section 3.12.)

Definition 3.2. A signature scheme DS is UF-CMA secure, if for any forger $\mathcal{F}$ running in time at most t, making at most q_h hash queries and making at most q_s signature queries, we have:

$$\mathbf{Adv}^{\text{UF-CMA}}_{\mathcal{F},\text{DS}} = \Pr\begin{bmatrix} 1 \leftarrow \mathbf{Finalise}(m^*, \sigma^*); \\ (m^*, \sigma^*) \leftarrow \mathcal{F}^{\mathbf{Hash}(\cdot),\mathbf{Sign}(\cdot)}(pk); \\ \texttt{pk} \xleftarrow{\$} \mathbf{Initialise}(1^\lambda) \end{bmatrix} \leq \varepsilon$$

3.3.2 Stronger Notion of Secure Signatures

While a secure digital signature ensures that an adversary cannot forge a signature for a new, previously unsigned message, it does not prevent the adversary from generating a new, valid signature for a message that has already been signed. To address this, a stronger notion of security is needed.

Consider a modified game `SUF-CMA(ROM)` defined in exactly the same way as Game `UF-CMA(ROM)`, except that now queries are to be restricted to an underlying Signature-Messages space $\mathcal{S}$. $\mathcal{S}$ instead contains pairs of oracle queries and their associated responses (e.g., $(m^*, \sigma^*) \in \mathcal{S}$ if $\mathcal{F}$ queried $\text{Sign}(m^*)$ and received in response the signature σ^*). The Sign and Finalise oracles

are adapted as follows:

Game SUF-CMA(ROM)

Sign(m)
$\sigma \xleftarrow{\$} \text{Sign}(sk, m)$
$\mathcal{S} \leftarrow \mathcal{S} \cup \{(m, \sigma)\}$
return σ

Finalise(m^*, σ^*)
if $\text{Vrfy}(pk, m^*, \sigma^*) == 1 \wedge (m^*, \sigma^*) \notin \mathcal{S}$
  return 1
else
  return 0

Figure 3.2: Game defining SUF-CMA

Definition 3.3. A signature scheme DS is said to be *Strong Existentially Unforgeable under an Adaptive Chosen-Message Attack* (SUF-CMA) secure if, for any forger $\mathcal{F}$ running in time at most t, making at most q_h hash queries and making at most q_s signature queries, we have:

$$\mathbf{Adv}^{\text{SUF-CMA}}_{\mathcal{F},\text{DS}} = \Pr\begin{bmatrix} 1 \leftarrow \mathbf{Finalise}(m^*, \sigma^*); \\ (m^*, \sigma^*) \leftarrow \mathcal{F}^{\mathbf{Hash}(\cdot),\mathbf{Sign}(\cdot)}(pk); \\ \texttt{pk} \xleftarrow{\$} \mathbf{Initialise}(1^\lambda) \end{bmatrix} \leq \varepsilon$$

3.4 RSA

RSA public key algorithm was developed by Ron Rivest, Adi Shamir and Leonard Adleman in 1977 [13]. Soon it became popular and was adopted by many standard bodies.

3.4.1 RSA Key Generation

RSA is widely used as the basis for digital signature schemes. There are various methods for generating digital signatures using RSA functions based on the RSA assumption.

While these methods differ in terms of operations in relation to signature and/or verification algorithms, the RSA key generation procedure is common to all RSA-based signature schemes. An RSA key consists of three elements: A modulus N, a public exponent e and a private exponent d. The pair (N, e) constitutes the public key while (N, d) forms the private key.

Definition 3.4. Let *GenModulus* be a polynomial-time algorithm that, that on inputs (1^λ, k, $(\lambda_1, ..., \lambda_k)$ outputs (N, $(p_1, ..., p_k)$) where $N = \prod_{i=1}^{k} p_i$ i.e., the product of k distinct random prime numbers $p_i \in_R \mathbb{P}[\lambda_i]$, for $i \in [\![1, \ldots, k]\!]$ for k constant.

To generate an RSA key pair, Each entity B runs the following algorithm:

Definition 3.5. GenRSA

1. Run GenModulus $(N, p_1, ..., p_k) \leftarrow GenModulus(1^\lambda, k, (\lambda_1, ..., \lambda_k))$
2. Compute $\phi(N) = \prod_{i=1}^{k}(p_i - 1)$
3. Select an arbitrary integer e, $e > 1$, such that gcd(e, $\phi(N)$) = 1
4. Compute d, $1 > d < \phi(N)$, such that $ed \equiv 1 \ (mod \ \phi(N))$
5. return N, e, d

The exponents are chosen in a way that for any number S with $S < N$, the following is always true:

$$S = M^{d \cdot e} \bmod N = M^{e \cdot d} \bmod N \ (1)$$

3.4.2 RSA Assumption

RSA's security is closely tied into the hardness of factoring with the concept of the RSA problem. Notably It's widely believed that if one could efficiently factor N into its prime components, they could solve the RSA problem. While the converse is not proven (if one could solve the RSA problem, this does not equate to factorising N efficiently), the link is strong enough - at least until the use of quantum computers becomes feasible.

Given a modulus N and a co-prime integer e, exponentiation to the eth power modulo N generates a permutation, leading to the RSA problem's core concept. For any number $y \in \mathbb{Z}_N^*$, if $x^e = y \mod N$, then x is uniquely defined, setting up the RSA problem to compute $x = [y^{1/e} \mod N]$, or the eth root modulo N, without the factorisation of N. Informally this is the RSA assumption.

Trapdoors and RSA. The strength of the RSA algorithm can be seen through the lens of trapdoor permutations. This process is easy to compute in one direction (finding y) but computationally hard in reverse (finding x), without the trapdoor d, which allows for the easy computation of the eth root modulo N.

Definition 3.6. (RSA Assumption [13]). The k-$\texttt{RSA}[\lambda]$, states that given (N, e, x^e) it is hard to compute x,

where N is a λ-bit number such that $N = \prod_{i=1}^{k} p_i$

i.e., the product of k distinct random prime numbers $p_i \in_R \mathbb{P}[\lambda_i]$, for $i \in [\![1, \ldots, k]\!]$, for k constant.

Additionally, $e \in \mathbb{Z}_{\phi(N)}^*$, and $x \in_R \mathbb{Z}_N$. k-$\texttt{RSA}[\lambda]$ is said to be (t, ε)-hard, if for all adversaries $\mathcal{A}$ running in time at most t, we have

$$\mathbf{Adv}_{\mathcal{A}}^{\text{k-RSA}[\lambda]} = \Pr[x = \mathcal{A}(N, e, x^e \bmod N)] \leq \varepsilon.$$

The assumption is equivalent to viewing of the RSA function as a trapdoor function. In turn it can be said that the security of any RSA-based signature scheme can be reduced to the security of the RSA function as a trapdoor function.

3.5 Textbook RSA

In its most basic form, RSA offers a clear blueprint for digital signatures. The loose resemblance between signatures and the RSA function hinges on their shared trait of asymmetry. While everyone should have the ability to verify a signature only the one possessing the signing key can have the capability to create a (legitimate) signature. The RSA function mirrors this asymmetry: If N and e are made public, then anyone can exponentiate using e ($m \stackrel{?}{=} [\sigma^e \bmod N]$), but only the individual with d can exponentiate using d ($\sigma := [m^d \bmod N]$).

Regrettably, textbook RSA signatures have security issues because the difficulty of RSA problem does not meaningfully relate to the computation of a signature, especially for non-uniform messages. Adversaries might bypass the RSA problem or deduce new signatures from others, leading to vulnerability.

Figure 3.3: Multiplicative property

$$(m_1 m_2)^d \equiv m_1^d m_2^d \bmod N \tag{3.1}$$

Multiplicative attacks leverage RSA's inherent mathematical property outlined more generally above: the product of two signatures is a valid signature for the product of their respective messages. More specifically, if σ_1 and σ_2 are respective signatures for m_1 and m_2, then $\sigma_1 \cdot \sigma_2$ is a valid signature for $m_1 \cdot m_2$. This allows forging by choosing messages to yield a desired product modulo N in the most severe form of attack.

3.6 Hash-and-Sign

Hash-and-Sign is designed to counteract the vulnerabilities observed in the textbook RSA signature schemes by enabling the disruption of algebraic relationships between plaintexts and ciphertexts. Secondly, it accounts for the fact that in real-world applications, messages are often bit strings of arbitrary lengths, not readily compatible with the format required for plain RSA signatures, which typically demand group elements.

To address these issues, a pre-processing step is commonly employed, which involves applying a transformation to a message x i.e., computing y := H(x) where H is a public hash function that outputs elements belonging to $\mathbb{Z}_N^*$.

Signatures are then computed on the resulting message representative (e.g., $\sigma := [y^d \bmod N]$) while verification now ensures signature equivalence with the corresponding representative (e.g., $\sigma^e \stackrel{?}{=} y \bmod N$). This significantly thwarts the efficacy of the discussed attacks if H is not efficiently invertible.

3.7 Classification of Digital Signature Schemes

Digital Signature schemes can be divided according to two general classes:

Basic Definition 1. *Digital signature schemes with appendix.* Digital signature schemes where the signature is appended to the message as a separate entity and subsequently both are required for verification. On receipt of the message and signature, the verifier uses the signer's public key to apply the relevant cryptographic operation to the signature, and compares the result to the hash of the message.

Basic Definition 2. *Digital signature schemes with message recovery.* Digital signature schemes where a portion of the message (full or partial) is embedded within the signature itself and the verification algorithm only requires a message portion if the full message was unable to be embedded in the signature. In cases of the latter, the portion of the message not embedded is transmitted along with the signature. On receipt of the signature and potentially a message (if applicable), the verifier uses the signer's public key to apply the relevant cryptographic operation to the signature, to recover the padded message and thereafter the message.

Digital signature schemes with message recovery are useful in scenarios with limited bandwidth or storage capacity, as they reduce the overall data size that needs to be transmitted and stored– for example smart cards. However, In the vast majority of applications this is unnecessary and signature schemes with appendix are typically the default choice.

3.8 RSASSA-PKCS1-v1.5

The PKCS#1 signature scheme is one of the earliest standardised Hash-and-Sign signature schemes on record. The scheme was initially disclosed in version 1.5 [1], which is why it is typically known as PKCS1 v1.5.

Messages are encoded in representative "blocks" in PKCS v1.5#1 with the (hexadecimal) format

$$0x00||BT||PS||0x00||D$$

This format starts with a leading 0x00 block, which serves to guarantee that the entire message representative, when converted to an integer, remains smaller than the modulus N. BT refers to the type of block which is 0x01 for signatures. 'D' represents the encoded message, consisting of the message's hash preceded by the hash identifier ID_H. 'PS' is the padding string, which is utilised to make sure that the message representative's length matches the bit length of the modulus and consists of a sequence of 0xFF bytes repeated as needed. Finally a trailing 0x00 block in the context of the padding serves as a separator from the encoded message 'D' that follows.

$$0x00||0x01||0xFF...FF||0x00||ID_H||H(m)$$

For all of the signature scheme definitions to follow: let GenRSA be adapted as to not compute and subsequently return d but instead return the prime factors of N as additional arguments.

RSA-PKCS1-v1.5 signatures

Gen$(1^\lambda, k, (\lambda_1, ..., \lambda_k), \ell)$

Run GenRSA$(1^\lambda, k, (\lambda_1, ..., \lambda_k))$ to obtain $(N, e, (p_1, ..., p_k))$

Choose a hash function $H : \{0,1\}^* \rightarrow \{0,1\}^\ell$.

Look-up α-bit ID_H for H

Compute $\nu = n - \ell - \alpha - 23$

Compute PAD $= 0^{15} \| 1^\nu \| 0^8 \| ID_H$

return $(pk = (N, e, PAD, H), sk = (p_1, ..., p_k))$

Sign(sk, m)

Compute $z \leftarrow H(m)$

Compute $y = PAD \| z$

return $\sigma = y^{1/e} \bmod N$

Vrfy(pk, m, σ)

Compute $y^{'} = \sigma^e \bmod N$

Compute $z \leftarrow H(m)$

If $(PAD \| z == y^{'})$

  return 1

else

  return 0

Figure 3.4: RSA PKCS#1 v1.5

Definition 3.7.

A java implementation of the schemes encoding is as follows:

```
1   @Override
2   protected byte[] encodeMessage(byte[] M) throws DataFormatException {
3
4     this.md.update(M);
5     byte[] mHash = this.md.digest();
6     byte[] digestInfo = createDigestInfo(mHash);
7     int tLen = digestInfo.length;
8
9     //Prepare padding string PS consisting of padding bytes (0xFF).
10    int psLength =
11        this.emLen - tLen - 3;
12    byte[] PS = new byte[psLength];
13    Arrays.fill(PS, (byte) 0xFF);
14
15    // Concatenate PS, D, and other padding to form the encoded message EM.
16    byte[] EM = new byte[emLen];
17    int offset = 0;
18    EM[offset++] = 0x00;
19    EM[offset++] = 0x01; // Block type 0x01 for PKCS signatures
```

```
20      System.arraycopy(PS, 0, EM, offset, psLength); // Padding
21      offset += psLength;
22      EM[offset++] = 0x00; // Separator
23      System.arraycopy(digestInfo, 0, EM, offset, tLen); // D
24
25      return EM;
26    }
```

The full class can be found at application/modules/signatures/models/schemes/RSASSA_PKCS1_v1_5.java.

3.9 ANSI X9.31 rDSA Signatures

ANSI X9.31 [14] is another Hash-and-Sign signature scheme standardised around a similar time to RSASSA-PKCS. Designed for use in the banking sector, the scheme is very similar to RSASSA-PKCS1-v1.5. Both variants of the signature schemes with appendix class of signatures, the two differ slightly with respect to padding and ordering of hash related data.

$$PS \| D$$

The format ends with a segment labeled 'D' representing the encoded message, which again includes a hash identifier and the message hash. Notably, the sequence here is inverted from the PKCS format: the message's hash is placed before the hash identifier. Before 'D', there's a padding string (PS) which is set to the sequence 0x6B...BA. Within this, 0xB...B represents the repetitive part of padding, which fills up the necessary space to ensure that the message representative's length matches the length of the modulus. An initial 0x6 nibble signifies the start of padding while a trailing 0x0A nibble indicates the end of padding.

$$0x6B...BA \| H(m) \| ID_H$$

It should be noted that the considered changes to padding are insignificant in the context of the proofs to follow, as they are both for arbitrary padding.

ANSI X9.31 rDSA signature scheme

Gen$(1^{\lambda}, k, (\lambda_1, ..., \lambda_k), \ell)$

Run GenRSA$(1^{\lambda}, k, (\lambda_1, ..., \lambda_k))$ to obtain $(N, e, (p_1, ..., p_k))$

Choose a hash function $H : \{0,1\}^* \rightarrow \{0,1\}^{\ell}$.

Look-up 16-bit ID_H for H

Compute $\nu = \dfrac{(\lambda - \ell - 16 - 8)}{4}$

Compute PAD $= 0110 \| (1011)^{\nu} \| 1010$

return $(pk = (N, e, PAD, ID_H, H), sk = (p_1, ..., p_k))$

Sign(sk, m)

Compute $z \leftarrow H(m)$

Compute $y = PAD \| z \| ID_H$

return $\sigma = y^{1/e} \bmod N$

Vrfy(pk, m, σ)

Compute $y^{'} = \sigma^{e} \bmod N$

Compute $z \leftarrow H(m)$

If $(PAD \| z == y^{'})$

  return 1

else

  return 0

Figure 3.5: ANSI X9.31 rDSA

Definition 3.8.

A java implementation of the schemes encoding is as follows:

```
1   @Override
2   protected byte[] encodeMessage(byte[] M) {
3     byte[] EM = new byte[emLen];
4     this.md.update(M);
5     byte[] mHash = this.md.digest();
6     byte[] digestInfo = createDigestInfo(mHash);
7     int tLen = digestInfo.length;
8     // copying the message hash to comprise the end
9     // of encoded message
10    int hashStart = emLen - tLen;
11    System.arraycopy(digestInfo, 0, EM, hashStart, tLen);
12    int delta = hashStart;
13
14    // Pad with Bs to fill the remaining space
15    if ((delta - 1) > 0) {
16      for (int i = delta - 1; i != 0; i--) {
17        EM[i] = (byte) 0xbb;
18      }
```

```
19        EM[delta - 1] ^= (byte) 0x01;
20        EM[0] = (byte) 0x6B;
21      } else {
22        EM[0] = (byte) 0x6A;
23      }
24      return EM;
25    }
```

The full class can be found at application/modules/signatures/models//schemes/ANSI_X9_31_RDSA.java.

3.10 ISO/IEC 9796-2:2010 Signature Scheme 1

The ISO/IEC 9796-2:2010 [15] Signature Scheme is the final standardised scheme to be considered. The scheme has widespread practical significance, primarily in the payments sector where it is used in the EMV (Europay, Mastercard, and Visa) payment system for chip and pin cards.

A variant of the signature schemes with message recovery class of signatures, ISO/IEC 9796-2:2010 offers a swift departure from the more common practice of signing with appendix. It differs from convention by embedding either the entire or part of a message within the signature. This allows for the recovery of the corresponding message representative upon verification. The approach yields space and message length savings, although it requires more computational effort to extract the message during verification.

EMV Protocol: The consideration of ISO/IEC 9796-2 Signature Scheme 1 is confined to its application within the EMV standard. Previous versions of the schemes general susceptibility to attacks [16], [17] is thus immaterial since such attacks work only on message spaces that exceed the length of the messages space used by the EMV protocol.

The standard offers two modes: full message recovery for sufficiently shorter messages that can be completely embedded in the signature and partial message recovery, which is applicable for extensively ranged messages that exceed the signature's capacity. As part of the latter, any message excess is transmitted along with the signature.

Table 3.1: ISO/IEC 9796-2 Signature block Format table

Full Message Recovery	$0x4B_0, ..., B_qA \| m \| H(m) \| 0xBC$
Partial Message Recovery	$0x6B_0, ..., B_qA \| m \| H(m) \| 0xBC$

The format of the message representative starts with an initial padding pattern to the left of message which can be denoted as PAD_L for illustration purposes. PAD_L starts with a header nibble that distinguishes between the two variants of the scheme with 0x6 for partial message recovery or alternatively 0x4 for full message recovery. The trailer nibble of 0xA is an indicator to the end of PAD_L.

In between PAD_L's header and the trailing 0x0A nibble is the repetitive portion of padding: an optional sequence of Bs, $B_0, ..., B_q$ where q is the number of bytes required to fill in the remaining space for the full message representative. This sequence is optional and is utilised only when the recoverable message following PAD_L is shorter than the available space, as seen in the full message recovery scheme. The first nibble in the B_q byte i.e., 0xB0 combines with the trailing 0x0A nibble to finalise PAD_L. After PAD_L is the recoverable message

portion which is followed by the hash of the complete message. The signatures conclude with a right padding pattern 0xBC denoted as PAD_R.

Typically, descriptions of the scheme do not mention the optional Bs and present PAD_L as solely comprising the header and trailer, i.e., $PAD_L = 0x4A$ or $PAD_L = 0x6A$.

*Note: For formal definitions to follow, for any sufficiently long bit string x, MSBs(x, n) denotes the n most significant bits of x.

ISO/IEC 9796-2:2010 Signature Scheme 1 with partial message recovery

Gen$(1^\lambda, k, (\lambda_1, ..., \lambda_k), \ell)$

Run GenRSA$(1^\lambda, k, (\lambda_1, ..., \lambda_k))$ to obtain $(N, e, (p_1, ..., p_k))$

Choose a hash function $H : \{0,1\}^* \rightarrow \{0,1\}^\ell$.

Compute $PAD_L = 01101010$

Compute $PAD_R = 10111100$

return $(pk = (N, e, PAD_L, PAD_R, H), sk = (p_1, ..., p_k))$

Sign(sk, m)

Compute $z \leftarrow H(m)$

Compute $\nu = \lambda - \ell - 16$

Compute $m_1 = \text{MSBs}(m, v)$ distinct from m such that $m = m_1 m_2$

Compute $y = PAD_L \| m_1 \| z \| PAD_R$

Compute $\sigma = y^{1/e} \bmod N$

return (σ, m_2)

Vrfy(pk, m_2, σ)

Compute $y' = \sigma^e \bmod N$ and parse y' as:

  $y' = PAD_L \| m_1 \| z \| PAD_R$

If $(H(m_1 \| m_2) == z)$

  return 1

else

  return 0

Figure 3.6: ISO/IEC 9796-2 Scheme 1 (PR)

Definition 3.9.

ISO/IEC 9796-2:2010 Signature Scheme 1 with full message recovery

Gen$(1^{\lambda}, k, (\lambda_1, ..., \lambda_k), \ell)$

Run GenRSA$(1^{\lambda}, k, (\lambda_1, ..., \lambda_k))$ to obtain $(N, e, (p_1, ..., p_k))$

Choose a hash function $H : \{0,1\}^* \rightarrow \{0,1\}^{\ell}$.

Compute $\nu = \dfrac{(\lambda - \ell - 8 - 4)}{4}$

Compute $PAD_L = 0100 \| (1011)^{\nu} \| 1010$

Compute $PAD_R = 10111100$

return $(pk = (N, e, PAD_L, PAD_R, H), sk = (p_1, ..., p_k))$

Sign(sk, m)

Compute $z \leftarrow H(m)$

Compute $y = PAD_L \| m \| z \| PAD_R$

return $\sigma = y^{1/e} \bmod N$

Vrfy(pk, σ)

Compute $y^{'} = \sigma^e \bmod N$ and parse $y^{'}$ as:

$y^{'} = PAD_L \| m \| z \| PAD_R$

If $(H(m) == z)$

return 1

else

return 0

Figure 3.7: ISO/IEC 9796-2 Scheme 1 (FR)

Definition 3.10.

A java implementation of the schemes encoding with the message recovery mode implicitly selected according to the length of chosen message (M) and modulus (emLen) is as follows:

```
1    @Override
2   public byte[] encodeMessage(byte[] M) throws DataFormatException {
3     byte[] EM = new byte[emLen];
4     m1Len = M.length;
5     int availableSpace = (hashSize + m1Len) * 8 + 8 + 4 - emBits;
6     //Partial recovery if message is larger than available space
7     // else scheme proceeds with full recovery.
8     if (availableSpace > 0) {
9       PADLFIRSTNIBBLE = 0x60;
10      isFullRecovery = false;
11    } else {
12      PADLFIRSTNIBBLE = 0x40;
13      isFullRecovery = true;
14    }
15
16    int hashStart = emLen - hashSize - 1;
17    int delta = hashStart;
```

```
18      //length of the message to be copied is either the availableSpace most
        significant bits of
19      // M or alternatively the full length of the original message if the
        message is too short
20      int messageLength = Math.min(m1Len, m1Len - ((availableSpace + 7) / 8) -
        1);
21      // m2 comprises the non-recoverable message portion
22      m2Len = max(m1Len - messageLength, 0);
23      //copying the message
24      delta -= messageLength;
25      System.arraycopy(M, 0, EM, delta, messageLength);
26
27      // Hash message as normal for standard case, else apply the variable
        length hash function
28      // to generate either the preset provable hash output (1/2 length of
        modulus) or any custom output size
29      byte[] hashedM = computeHashWithOptionalMasking(M);
30      System.arraycopy(hashedM, 0, EM, hashStart, hashSize);
31
32      // Pad with Bs if m_r (m1) is shorter than the available space
33      if ((delta - 1) > 0) {
34        for (int i = delta - 1; i != 0; i--) {
35          EM[i] = (byte) 0xbb;
36        }
37        // The case of full recovery:
38        // modify the second nibble of final PAD_L 0xBB byte to
39        // contain 0x0A as per the scheme
40        EM[delta - 1] ^= (byte) 0x01;
41        EM[0] = (byte) 0x0b;
42        EM[0] |= PADLFIRSTNIBBLE;
43      } else {
44        //The case of partial recovery: no B bytes required
45        // So update the second nibble of final and first PAD_L byte
46        // to contain 0x0A as per the scheme
47        EM[0] = (byte) 0x0a;
48        EM[0] |= PADLFIRSTNIBBLE;
49      }
50      EM[emLen - 1] = PADR;
51      return EM;
52    }
```

The full class can be found at application/modules/signatures/models//schemes/ISO_IEC_9796_2_SCHEME.

3.11 Motivation for Provable Security

The hash-and-sign paradigm shows potential in countering known attacks by enabling the admission of a function H not efficiently invertible. However this is no replacement for formal proof. This is especially evident considering the Multiplicative property that hash-and-sign methods inherit from textbook RSA. It becomes unfeasible to imagine how $H(m_1) \cdot H(m_2) \bmod N$ could have the algebraic structure required to make it look like the hash of some other distinct message m. While at the very most a solution to this problem is not

evident, this is not a proof that the attack is impossible and thereby immaterial.

For a provably secure signature in the UF-CMA sense, a Hash function H that avoids multiplicative relations is needed. Moreover a stronger assumption to make would be: it must be hard to find collisions in H. Currently, there's no way to ensure the basic hashed RSA signature scheme is provably secure in standard settings.

The Hash-and-Sign scheme focuses on countering known threats and is based on classical design principles. While identifying essential features of the hash function offers some insight, it's just a starting point. Over-reliance on known threats is risky, as unforeseen flaws can exist due to unidentified features, potentially compromising security.

A better strategy is to understand how a signature scheme's security relates to the underlying primitive's assumed security, like the RSA Assumption. The focus should extend beyond just countering known threats to addressing all potential threats, even unknown ones. Under this very approach the (basic) hashed RSA signature scheme becomes unacceptable. This related philosophy drives the concept of provable security.

3.11.1 The difficulty of proving security of RSA PKCS#1 v1.5 signatures

Provable security requires linking the security of a signature scheme to the assurance of its underlying primitives. For RSA signatures, this necessitates tying the security of the signature scheme to the hardness of the RSA problem, and some other security condition (collision resistance) on the hash function.

A complication arises with the PKCS#1 v1.5 signature scheme, stemming from potential message representatives described by the set

$$S_N = \{(PAD\|z) \mid \sigma = (PAD\|z)^{1/e} \bmod N\}.$$

Given a hash function like SHA-1 yielding l bits, the magnitude of S_N is bounded by 2^l. For example with SHA-1, this would mean $|S_N| \leq 2^{160}$. Despite its seeming breadth, S_N is insignificant within the RSA domain Z_N^*, insinuating a negligible chance for random elements from Z_N^* to belong to S_N.

This highlights a concern: it becomes feasible to compute e-th roots for values within S_N. This differential hardness poses a challenge. If the RSA function can be easily inverted on S_N, then signatures can be forged, and the security of the PKCS signature scheme can be compromised. Hence, relying simply on the RSA assumption does not directly ensure the security of the PKCS signature scheme.

The vulnerability is not necessarily in the hash function but in how the message, after hashing, is combined with deterministic padding to land in S_N. Because S_N lacks a known algebraic structure, conventional proof methods do not work. The deterministic nature of S_N might expose the scheme to potential attacks, making the subset's predictability a risk. To ensure security, the RSA assumption would also need to apply more specifically to S_N. Even then, this is merely a necessary condition, not a guarantee of the signature scheme's security.

In summary, the lack of known attacks does not guarantee the security of the PKCS scheme or any cryptographic system. For the signature schemes examined in this project, design flaws with RSA usage can be pinpointed. Such flaws deviate from the understanding of the security of RSA based on its accompanying problem and this is cause for concern.

3.11.2 Limitations of Provable Security

As central as the paradigm of Provable Security is, it is important to keep in mind some of its limitations. It does not entail security absolute sense; rather, it depends on specific assumptions. In practice, actual security might differ from what is provable due to real-world vulnerabilities not covered in theoretical models. This means that security claims may not fully encompass real-world scenarios and a cryptographic primitive deemed theoretically unbreakable may still have unrelated vulnerabilities when put into practice. To formulate security definitions that provide meaningful guarantees, it is thus necessary to understand how primitives are used in practice.

For instance, while indistinguishability theory posits that encrypted plaintexts of the same length should be indiscernible, real-world scenarios might allow deductions from ciphertext lengths, potentially compromising confidentiality. Understanding the security that a proof provides for a specific cryptographic scheme is different from understanding the scheme overall security requirements. Often, real-life applications demand security requirements that exceed what theoretical proofs cover. Moreover, the adversary models used in theoretical proofs may not fully capture the abilities of genuine adversaries. Attacks like padding oracle attacks, where adversaries glean plaintext information, are not always accounted for in security models. Similarly, side-channel attacks exploit external information beyond a model's scope. The relevance of a theoretical model hinges on how closely it simulates the strategies of real attackers.

Implementing cryptographic systems is intricate, with design, integration, and coding all introducing potential pitfalls. While specifications may offer some leeway for those implementing the system, lacking clear and thorough instructions could lead them to make decisions that significantly compromise the security. Even a slight tweak to a proven system can drastically compromise its security. Thus, it's paramount that real-world applications adhere to their theoretical models as closely as possible.

3.11.3 Benefits and Real World Implications

The ad hoc security approach, relying on attack obscurity, is not only flawed but misleading, as seen in PKCS's encryption scheme within the PKCS#1 V1.5 standard from 1993 [1]. Despite rigorous scrutiny, schemes once thought secure were eventually breached. This paved the way for provable security, emphasising concrete security principles and the quality of cryptographic proofs.

By 1998, Bleichenbacher showcased an adaptive chosen-ciphertext attack on the PKCS#1 v1.5 encryption scheme [9]. Here, the decryption determined plaintext's validity based on padding. With incorrect padding, an error message was returned. This was exploited in some SSL/TLS protocol implementations, allowing adversaries to decipher plaintexts from error message information. This attack had profound implications, including a full class subsequent attacks, even up to two decades later (e.g., [18]–[27]).

The reach of Bleichenbacher's attack extended to incorrect implementations of the PKCS signature scheme [10], [28], enabling digital signature forgery and even more devastatingly, potentially allows for the impersonation of a vulnerable server without even requiring the respective private key. A real-world example targeting Facebook [27] in 2018 underscored the risk. Though Meta since rectified their specific TLS implementation, the issue was widespread, affecting almost a third of the top 100 domains in the Alexa Top 1 Million list, and additionally many major domains and products from key vendors and open-source initiatives.

Other versions of the attack exploited code flaws in non-patched OpenSSL versions [29]. Some attacks bridged protocols and versions, targeting servers indirectly connected to vulnerable ones, highlighting the challenges of implementing RSA with PKCS#1 v1.5 padding. These examples underscores the pitfalls of an ad-hoc security approach and the necessity for rigorous, provable security measures to ensure cryptographic schemes can withstand evolving threats over time.

3.12 Random Oracle Model

The complexity of finding proofs for Hash-and-Sign schemes has been discussed but ultimately there are now proofs that apply in the random oracle model (ROM). The ROM [30] is essentially an idealised theoretical model providing access to a hash function H, which is treated as a blackbox (for example, by calling it as a subroutine or by communicating with another computer program). This in turn allows you to prove a protocol to be correct assuming that H maps each input to a truly random output, i.e., it behaves like a truly random oracle. The oracle also "remembers" all inputs and if the same input is given, it produces the same output.

It can be said the hash function is "full-domain" because its values range through the full interval of the modulus i.e., [0, N] enabling messages to be hashed in a way that makes the resulting points more evenly distributed across the full RSA domain. This is an effective mitigation acting as resolution to the basic hash issue of messages being clumped in a tiny and predictable subset. Although truly random functions cannot be implemented in reality, the resulting soundness the ROM facilitates in a scheme's design allows some measure of confidence to derived at the very least. Loosely it provides a guarantee that a scheme is not flawed, based on the intuition that an attacker would be forced to use the hash function in a non generic way. This technique is the starting point, providing the foundational setting for the proof of deterministic signature schemes.

Chapter 4: **Security Proofs**

Due to the breakthrough in 2018 when a security proof [3] was provided for RSASSA-PKCS1-v1.5. It is possible to formally prove the security of the full class of deterministic RSA signatures. The precise statement of security is if the RSA problem is hard when H is modelled as a random oracle, then the scheme is secure (with regards to the UF-CMA notion i.e., existential unforgability against adaptive chosen message attacks). There are two different proofs for this, both which consider case where the modulus $\widehat{N}$ is a product of three primes. The first is based on the RSA assumption, the second is based on the ϕ-Hiding assumption.

4.1 Encode Algorithm

Jager, Kakvi and May overcame the difficulty of providing proof for PKCS with a specialised encode algorithm that allows the simulation of signatures in polynomial time.

Definition 4.1. (*Encode* [3]). Let

$$(\widehat{y}, s, z) \xleftarrow{\$} \text{Encode}(N, e, y, \ell, \text{PAD}, R)$$

be an efficient algorithm that takes as input an n-bit integer N, an exponent e, $y \in \mathbb{Z}_N^*$, a hash value length ℓ, padding pattern PAD and an r-bit prime $R \in \mathbb{P}[r]$, and outputs $(\widehat{y}, s, z) \in \mathbb{Z}_{\widehat{N}} \times \mathbb{Z}_N^* \times \{0,1\}^\ell$ or failure $\perp$.

The algorithm outputs $(\widehat{y}, s, z)$ such that $\widehat{y}$ has to be correct form for a PKCS#1 signature mod $\widehat{N}$ where z constitutes message hash and s comprises an eth root mod N. Using this and the knowledge of R, the eth root modulo $\widehat{N}$ can be computed. More precisely it enables the encoding of an arbitrary integer y modulo N as an integer $\hat{y}$ modulo $\hat{N} = NR$, for some prime R, such that $\widehat{y}$ has correct PKCS#1-V1.5 padding modulo $\hat{N}$.

1. y denotes an embedded RSA challenge that given a forgery can be solved i.e., obtaining $\widehat{y} = PAD \| \| z$.
2. In the case that y is replaced by 1 the algorithm instead stimulates a signature.

With these two uses of the Encode algorithm, UF-CMA security of PKCS#1 was proved in the ROM.

Encode$(N, e, y, \ell, \text{PAD}, R)$

$n = \lceil \log_2 N \rceil\ ,\ r = \lceil \log_2 R \rceil$

$z := 2^{\ell},\ k := 0$

while $(z \geq 2^{\ell})$ **and** $(k < n \cdot 2^{-\ell})$:

  $k := k + 1$

  $s \xleftarrow{\$} \mathbb{Z}_N$

  $z := y^s e - 2^{\ell} \cdot \text{PAD} \mod N$

  $\widehat{y} := 2^{\ell} \cdot \text{PAD} + z$

if $z < 2^{\ell}$ **then**

  return $(\widehat{y}, s, z)$

else

  return $\perp$.

Figure 4.1: Encode algorithm [3]

The algorithm outputs $(\widehat{y}, s, z)$ except with negligible failure probability (in n). It can be said that encode efficiently $(n - \mathcal{O}(log\ n))$-simulates the PKCS#1 v1.5 encoding modulo $\hat{N} = NR$ which is true for a large hash value $\ell \approx |n|$.

4.2 Background

See section 3.4.2 for the intuition behind trapdoor permutations in the context of its application to RSA.

Definition 4.2. A *family of trapdoor permutations* [31]. $TDP = (\text{Gen}, \text{Eval}, \text{Invert})$ comprises the following three polynomial-time algorithms:

1. **Gen:** A probabilistic algorithm that, given input 1^{λ}, produces a public description pub (inclusive of an efficiently sampleable domain Dom_{pub}) and a trapdoor td.
2. **Eval:** A deterministic algorithm that, given pub and $x \in \text{Dom}_{\text{pub}}$, yields $y \in \text{Dom}_{\text{pub}}$. This relationship is expressed as $f(x) = \text{Eval}(\text{pub}, x)$.
3. **Invert:** A deterministic algorithm that, given td and $y \in \text{Dom}_{\text{pub}}$, produces $x \in \text{Dom}_{\text{pub}}$. This relationship is described by $f^{-1}(y) = \text{Invert}(\text{pub}, y)$.

It is required for all $\lambda \in \mathbb{N}$ and any (pub, td) produced by $\text{Gen}(1^{\lambda})$:

$$f(\cdot) = \text{Eval}(\text{pub}, \cdot) \text{ must define a permutation over } \text{Dom}_{\text{pub}}$$

and additionally for all $x \in \text{Dom}_{\text{pub}}$:

$$\text{Invert}(td, \text{Eval}(\text{pub}, x)) \text{ should equal } x$$

It's important to note that $f_{\text{pub}}(\cdot) = \text{Eval}(\text{pub}, \cdot)$ needs to be a permutation for a correctly generated pub.

A trapdoor permutation is certified if one can publicly verify that it actually defines a permutation.

Definition 4.3. *Certified Trapdoor permutation* [32]–[34]. A family of trapdoor permutations TDP is called *certified* if there exists a deterministic polynomial-time algorithm Certify that, on input of 1^λ and an arbitrary (polynomially bounded) bit-string pub (potentially not generated by Gen), returns 1 iff $f(\cdot) = \text{Eval}(\text{pub}, \cdot)$ defines a permutation over Dom_{pub}.

Lossy Trapdoor Permutations are a realisation of the lossiness security notion for trapdoor permutations. Essentially, these permutations function in two distinct modes. The first allows for complete input recovery using an (injective) trapdoor function, while the second ((lossy) trapdoor function) causes substantial input data loss. Notably, distinguishing between these two behaviours is hard for any efficient adversary.

Definition 4.4. *Lossy trapdoor permutation* [11], [35]. Let $l \geq 2$. A trapdoor permutation TDP is a (l, t, ε) *lossy trapdoor permutation* if the following two conditions hold:

1. There exists a probabilistic polynomial-time algorithm LossyGen, which on input 1^λ outputs pub$'$ such that the range of $f_{\text{pub}'}(\cdot) := \text{Eval}(\text{pub}', \cdot)$ under $\text{Dom}_{\text{pub}'}$ is at least a factor of l smaller than the domain $\text{Dom}_{\text{pub}'}$:

$$\frac{\text{Dom}_{\text{pub}'}}{f_{\text{pub}'}(\text{Dom}_{\text{pub}'})} \geq l$$

2. All distinguishers $\mathcal{D}$ running in time at most t have an advantage $\text{Adv}^L_{\text{TDP}}(\mathcal{D})$ of at most ε, where

$$\mathbf{Adv}^L_{\text{TDP}}(\mathcal{D}) = \Pr[\mathbf{L}^{\mathcal{D}}_1 \Rightarrow 1] - \Pr[\mathbf{L}^{\mathcal{D}}_0 \Rightarrow 1]$$

procedure Initialise Game L_0

$(pub, td) \xleftarrow{\$} \text{Gen}(1^\lambda)$

return pub

procedure Initialise Game L_1

$(pub', L) \xleftarrow{\$} \text{LossyGen}(1^\lambda)$

return pub'

Figure 4.2: The Lossy Trapdoor Permutation Games.

Definition 4.5. *Regular lossiness* [11], [35]. A TDP is regular (l, t, ε) lossy if the TDP is (l, t, ε) lossy and all functions $f_{\text{pub}'}(\cdot) = \text{Eval}(\text{pub}', \cdot)$ generated by LossyGen are l-to-1 on $\text{Dom}_{\text{pub}'}$.

Definition 4.6. *The RSA trapdoor permutation* [11]. `RSA` = (`RSAGen`, `RSAEval`, `RSAInv`):

1. `RSAGen`(1^λ) outputs `pub` = (N,e) and `td` = d, where $N = pq$ is the product of two $\lambda/2$-bit primes, `gcd`(e,$\phi(N)$) = 1, and $d = e^{-1} \mod \phi(N)$.
   - The domain is $\text{Dom}_{\texttt{pub}} = \mathbb{Z}^*_N$.
2. `RSAEval`(`pub`, x) returns $f_{\texttt{pub}}(x) = x^e \mod N$,

3. RSAInv(td, y) returns $f_{\mathtt{pub}}^{-1}(y) = y^d \mod N$.

In the context of RSA, lossy trapdoor permutations can be viewed as the converse of certified trapdoor permutations. For example they entail an impossibility to differentiate an honestly generated (N, e) from (N, $\hat{e}_{loss}$) for which $RSA_{N,\hat{e}_{loss}}$ is many-to-1, thereby disqualifying themselves as permutations.

4.2.1 2v3PA

A final step computation assumption relevant to security statements in both proofs is the 2 vs 3 primes assumption 2v3PA[λ]. It postulates that it's hard to discern if a specific modulus is derived from two or three prime factors. Although the assumption has never been formally studied it is widely accepted. Its role is to bridge the proofs to the setting of the typical two prime factor modulus.

Definition 4.7. *The 2 vs. 3 Primes Assumption* [3] . The 2v3PA[λ] states that it is hard to distinguish between N_2 and N_3, where N_2, N_3 are λ-bit numbers, where

1. $N_2 = p_1 p_2$ is the product of 2 distinct random prime numbers $p_1, p_2 \in \mathbb{P}$;
2. $N_3 = q_1 q_2 q_3$ is the product of 3 distinct random prime numbers $q_1, q_2, q_3 \in \mathbb{P}$

2v3PA[λ] is said to be (t, ϵ)-hard, if for all distinguishers $\mathcal{D}$ running in time at most t, we have:

$$\mathbf{Adv}_{\mathcal{D}}^{\mathtt{2v3PA}[\lambda]} = \Pr[1 \leftarrow \mathcal{D}(N_2)] - \Pr[1 \leftarrow \mathcal{D}(N_3)] \leqslant \epsilon$$

4.3 Proof Theorems

Both proofs consider signatures of the form

$$\sigma = (\gamma \cdot H(m) + f(m))^{1/2}$$

i.e., a scheme with (potentially) message-dependent padding. This stems from theorem statements used initially by Coron's proof [8] for the Rabin-Williams variant ($e = 2$) of RSASSA-PKCS1-v1.5. Coron's theorem statements were general enough to indeed be extended to the ANSI X9.31 rDSA and ISO/IEC 9796-2 Scheme 1 signatures. Therefore while not explicitly stated, it can be said that the considered proofs, essentially descendants of Coron's proof, also apply to the latter two schemes, with relevant adjustments.

Signature Scheme	**Message Representative**	γ **value**	$f(m)$ **Value**
PKCS#1 v1.5	$y = PAD \| z$	1	$\mathtt{PAD} \times 2^{\ell}$
ANSI X9.31 rDSA	$y = PAD \| z \| ID_H$	2^{16}	$\mathtt{PAD} \times 2^{\ell+16} + \mathtt{ID_H}$
ISO/IEC 9796-2 Scheme 1	$y = PAD_L \| m_1 \| z \| PAD_R$	2^8	$(\mathtt{PAD_L} \| m_1 \times 2^{\ell+8}) + \mathtt{PAD_R}$

Table 4.1: Signature Schemes considered in the context of Coron's proof [8]

4.3.1 Security Proof under the RSA Assumption

The computation assumption relevant to security statements in the first proof is the RSA Assumption k-RSA[λ] (See definition 3.6).

Theorem 1. (Jager-Kakvi-May [3]). Assume that 2-RSA[λ] is (t', ε')-hard. Then for any (q_h, q_s), RSASSA-PKCS1-v1.5 is $(t, \varepsilon, q_h, q_s)$-UF-CMA secure in the Random Oracle Model, where

$$\varepsilon' = \frac{\varepsilon}{q_s} \cdot \left(1 - \frac{1}{q_s + 1}\right) \approx \frac{\varepsilon}{q_s} \cdot \exp(-1)$$

$$t' = t + \mathcal{O}(q_h \cdot \lambda^4).$$

The proof requires the following adaptations to bounds for applicability to the remaining schemes.

Table 4.2: Adaptation of the 2-RSA[λ] proof bounds to ANSI and ISO Schemes

Signature Scheme	γ **value**	**Modified** t' **bound**
ANSI X9.31 rDSA	2^{16}	$t' = t + 2^{15} \cdot \mathcal{O}(q_h \cdot \lambda^4)$
ISO/IEC 9796-2 Scheme 1	2^8	$t' = t + 2^7 \cdot \mathcal{O}(q_h \cdot \lambda^4)$

The reductions used in the first proof bounds the probability ε of breaking `RSASSA-PKCS-V1.5` in time t by $\varepsilon' \cdot q_s$, where ε' is the probability of inverting RSA in time $t' \approx t$ and q_s is the number of signature queries by the forger. Equivalently, the security reduction is loose with a loss in the order of q_s. This is not ideal since it has a negative impact on the practical parameter choices for instantiations.

A realisation of this is the necessity for a significantly larger modulus relative to what can be achieved with a tight reduction to achieve a specific level of security, or more generally, obtaining a meaningful security proof. The loss is optimal for RSA with "large" exponents ([36], [11]). More precisely this refers to an exponent $e > N$ that is additionally prime [37], [38]. This is due to the fact that such an exponent defines RSA as a Certified Trapdoor Permutation. This is because if e is a prime, then it can never divide $\phi(N) < N$ and hence $\gcd(e, \phi(N)) = 1$.

In this case the 2-RSA[λ] proof implies SUF-CMA security as well as resilience against subversion attacks [12]. However this discovery serves primarily as a theoretical/initial insight and design validation since choosing $e > N$ is usually avoided in practice due to the costs for modular exponentiation.

4.3.2 Security Proof under the Phi-Hiding Assumption

The computation assumption relevant to security statements in the second proof is the φ-Hiding Assumption k-ϕHA[λ]. It posits that for a given modulus N and a sufficiently small exponent e ($e < N^{\frac{1}{4}}$), determining if $\frac{\phi(N)}{e}$ is hard.

By definition 4.5 when $\gcd(\hat{e}_{\text{loss}}, \phi(N)) = \hat{e}_{\text{loss}}$ the $RSA_{N,\hat{e}_{loss}}$ function is $\hat{e}_{loss}$-to-1 i.e., $\hat{e}_{\text{loss}}-$regular lossy.

Definition 4.8. *The φ-Hiding Assumption* [37]. The k-ϕHA[λ], states that it is hard to distinguish between (N, e_{inj}) and $(N, \hat{e}_{loss})$, where

1. N is a λ-bit number such that $N = \prod_{i=1}^{k} p_i$ i.e., the product of k distinct random prime numbers $p_i \in_R \mathbb{P}[\lambda_i]$, for $i \in [\![1, \ldots, k]\!]$, for k constant.
2. $e_{inj}, \hat{e}_{loss} > 3 \in \mathbb{P}$;

3. $e_{inj}, \hat{e}_{loss} \leq N^{1/4}$, with $\gcd(e_{inj}, \varphi(N)) = 1$ and $\gcd(\hat{e}_{loss}, \varphi(N)) = \hat{e}_{loss}$, where φ is the Euler Totient function.

k-ϕHA[λ] is said to be (t, ϵ)-hard, if for all distinguishers $\mathcal{D}$ running in time at most t, we have:

$$\mathbf{Adv}_{\mathcal{D}}^{\text{k-}\phi\text{HA}[\lambda]} = \Pr[1 \leftarrow D(N, e_{inj})] - \Pr[1 \leftarrow D(N, \hat{e}_{loss})] \leqslant \epsilon$$

Theorem 2. (Jager-Kakvi-May [3]). Assume 2-ΦHA[λ] is (t', ε')-hard and gives an η-regular lossy trapdoor function. Then, for any (q_h, q_s), RSASSA-PKCS1-v1_5 is $(t, \varepsilon, q_h, q_s)$-UF-CMA secure in the Random Oracle Model, where

$$\varepsilon = \left(\frac{2\eta - 1}{\eta - 1}\right) \cdot \varepsilon'$$

$$t = t' + \mathcal{O}(q_h \cdot \lambda^4)$$

The proof requires the following adaptations to bounds for applicability to the remaining schemes.

Table 4.3: Adaptation of the 2-RSA[λ] proof bounds to ANSI and ISO Schemes

Signature Scheme	γ **value**	**Modified** t **bound**
ANSI X9.31 rDSA	2^{16}	$t = t' + 2^{15} \cdot \mathcal{O}(q_h \cdot \lambda^4)$
ISO/IEC 9796-2 Scheme 1	2^8	$t = t' + 2^7 \cdot \mathcal{O}(q_h \cdot \lambda^4)$

The second proof bypasses the optimality result of the 2-RSA[λ] proof (i.e., tight security proof that is independent of q_s), by using "small" keys that do not define a certified trapdoor permutation. More precisely "small" keys, entail an exponent $e \leqslant N^{1/4}$ (i.e., e is at most $\frac{1}{4}$ of the bit-length of N). This allows for a security proof tight to the φ-Hiding assumption. Additionally, there is currently no known proof technique which achieves tighter security for small exponents.

The condition that $e_{inj}, \hat{e}_{loss} \leq N^{\frac{1}{4}}$ arises because with only the modulus N known, it is not possible to determine the number of prime factors it has or their relative sizes. It is understood that N is composite, implying it has a minimum of two prime factors. The bound then follows as a direct consequence of known attacks described by Coppersmith [39].

Such exponents define RSA as a Lossy Trapdoor Permutation. This is because when $\gcd(\hat{e}_{\text{loss}}, \phi(N)) = \hat{e}_{\text{loss}}$ the $RSA_{N,\hat{e}_{loss}}$ function is $\hat{e}_{loss}$-to-1 i.e., $\hat{e}_{\text{loss}}-$ regular lossy (see Definition 4.5). In this case the 2-ΦHA[λ] proof implies only the weaker UF-CMA security and no provable security against subversion attacks. The proof technique has direct relevance to real-world instantiations often in which small exponents e.g., $e = 2^{16} + 1$ are used.

4.4 Implications for practical instantiation

4.4.1 Both Proofs

Considering PKCS#1 v1.5 signatures instantiated with an λ-bit modulus:

Implication 1. *Hash output* $|H()| \geq \dfrac{\lambda}{2}$. Both proofs work only with a hash function that has an uncommonly large output. More precisely they work under the assumption that

breaking the RSA (or φ-Hiding) assumption is hard with respect to an $\frac{\lambda}{2}$-bit modulus N with $|N| = |H()|$. Concretely, for signatures instantiated with an 1024-bit RSA modulus $\hat{N}$, this would mean security in the context of a 512-bit RSA assumption, with 512-bit padding and a 512-bit hash function.

The PKCS#1 v2.2 standard, includes in Appendix B of RFC 8017 [6], a method for transforming a cryptographic hash function to generate outputs of any desired size. This method, known as Mask Generation Function 1 (MGF1) involves the recurrent application of a conventional hash function to the input alongside incrementally changing counter values to achieve the required output length.

Alternatively, the NIST FIPS 202 standard [40] specifies extendable-output functions (XOFs), called SHAKE-128 and SHAKE-256 under the SHA-3 umbrella, that have arbitrary length output. The method forming the basis for this is known as the Keccak construction [41]. It involves two phases: absorption, where input is padded, divided, and XORed with a state, and squeezing, where the state is repeatedly permuted to generate output of desired length.

Implication 2. *3-Prime Factor Modulus N.* The Encode algorithm requires the modulus to double in bit length when compared to the norm with a newly introduced third prime factor necessitating the increase in bits. Thus if a λ-bit modulus is used in the key for the security reductions of 2-ϕHA[λ] or 2-RSA[λ], this includes an additional $\frac{\lambda}{2}$-bits made up by a third prime factor. In turn the security proofs apply for keys where N $= p_1p_2p_3$, i.e., is the product of three primes. Under the assumption that 3-prime moduli are indistinguishable from 2-prime moduli, results can be brought back to the case N $= pq$.

4.4.2 Security Proof under the RSA Assumption

Implication 3. *arbitrary-e.* The proof allows for the selection of an arbitrary-e. Although the general security of a practical instantiation should be considered with respect to the aforementioned security loss (the number of signature queries) which is as discussed is not optimal in this setting.

4.4.3 Security Proof under the Phi-Hiding Assumption

Implication 4. *prime* $e \leqslant 2^{\frac{\lambda}{4}}$. The proof allows for the selection of a small e. This is ideal, since a widely-employed strategy, in order to enable efficient verification of signatures is to select small and fixed specific numbers such as $e = 3$ or $e = 2^{16} + 1$ (both of which $\leq N^{\frac{1}{4}}$). With such a choice, RSA signature verification is much faster than RSA signature generation which is useful because practically signature verification is often the predominant operation being performed.

4.4.4 Summary of Implications

Table 4.4: Parameter sizes and security of deterministic RSA hash-and-sign signatures (instantiated with a λ-bit modulus) [42]

Scheme	Proof methodology	Assumption	No. prime factors	Exponent e	$\lvert H(\cdot)\rvert$	Security loss
• RSASSA-PKCS1-v1_5 • ANSI X9.31 rDSA • ISO/IEC 9796-2 Scheme 1	Jager-Kakvi-May	2-RSA$[\lambda/2]$	3	Arbitrary	$\geqslant \lambda/2$	q_s
		$+2v$3PA$[\lambda]$	2	Arbitrary	$\geqslant \lambda/2$	q_s
	Jager-Kakvi-May	2-ϕHA$[\lambda/2]$	3	Prime $\leqslant 2^{\lambda/4}$	$\geqslant \lambda/2$	$\mathcal{O}(1)$
		$+2v$3PA$[\lambda]$	2	Prime $\leqslant 2^{\lambda/4}$	$\geqslant \lambda/2$	$\mathcal{O}(1)$

Chapter 5: **Practical Assumptions for Project Implementation**

5.1 Notion of a Provably secure Key Configuration

There are two potential impacts in relation to the choice of the public exponent e. The second one as per implication 4 of the 2-ϕHA[λ] proof and its associated compromise (tight proof independent of the number signature queries, but implies only the weaker UF-CMA-security, as opposed to the stronger SUF-CMA security obtained from the 2-RSA[λ] proof) was the one adopted in this project. Consequently, the following key configuration types are defined:

Basic Definition 3. *Provably secure key configuration.* 2 ($\lambda/2$, $\lambda/2$) or 3 ($\lambda/4$, $\lambda/4$, $\lambda/2$) prime factor modulus encompassed as the key size λ for a key generated with the selection of a small e value.

Basic Definition 4. *Standard key configuration.* 2 ($\lambda/2$, $\lambda/2$) or 3 ($\lambda/4$, $\lambda/4$, $\lambda/2$) prime factor modulus encompassed as the key size λ for a key generated with the selection of an arbitrary e value.

5.2 Notion of a Provably secure instantiation of a signature scheme

As detailed in Section 6.5, employing a hash output of at least $\frac{\lambda}{2}$ is essential for provable security in signature schemes. However, this enhanced hash output is not the sole requirement for a scheme's provable security. It must be complemented with a provably secure key configuration as defined in Definition 3. A signature scheme is thus considered provably secure when it combines both a large hash output and a provablt secure key configuration. This leads to the definition of two primary parameter types referenced throughour this report:

Basic Definition 5. *Instantiation of a scheme with standard parameters.* A scheme initialised using a λ-bit key generated using 2 or 3 prime factors and arbitrary e value configured to use a fixed size hash function e.g., a commonly used option is SHA-256.

Basic Definition 6. *Instantiation of a scheme with provably secure parameters.* A scheme initialised using a λ-bit key generated using 2 or 3 prime factors and small e value configured to use a hash function with an *output* $|H()| \geq \frac{\lambda}{2}$.

As with key configuration, the utilisation of both 2 and 3 prime factor setups extends to the general parameter types used to classify the instantiation of schemes. It is important to note that while a provably secure hash output could be any anything $\geq \frac{\lambda}{2}$ for the purposes of analysis in this project it is fixed to the preset proportion $\frac{\lambda}{2}$.

Chapter 6: **Main Deliverable**

*Appendix A presents my approach to software development and a discussion of technology choices relevant to implementation detail.

6.1 Requirements Specification

The overall goal of the system is to provide a benchmarking tool designed to quantify the computational time impact of implementing larger, provably secure parameters in the full suite of deterministic RSA signature schemes (albeit with a main focus on the PKCS standard), as opposed to the standard parameters commonly used. This program will be equipped with functionalities that allow users to tailor where relevant the parameters to their specific requirements before starting any benchmarking activities on large scale batches of data. Benchmarking activities will span the full digital signature workflow from key generation and signature creation, to signature verification.

6.1.1 Description of Actors

Table 6.1: Description of Actors

Actor / Role Name	**Role Description and Objective**
User/Signer	The individual in this role seeks to benchmark the signing process. For Key Generation, they select the number of keys, configure key sizes (e.g., 256, 256, 512) with an optional selection of small e for each, and start key generation benchmarking to run for a number of trials For signature creation, they supply a message batch, key batch, hash function, and launch benchmarking for their desired signature scheme and. The aim is to assess the efficiency of different signing configurations, including provably secure parameters.
Verifier	Responsible for benchmarking the verification process. The verifier provides a batch of signed messages and public keys, chooses a hash function, and starts verification benchmarking. The objective is to understand the verification efficiency of selected signature schemes with different parameter types, including standard and provably secure parameters.

6.1.2 Use Cases (Benchmarking Mode)

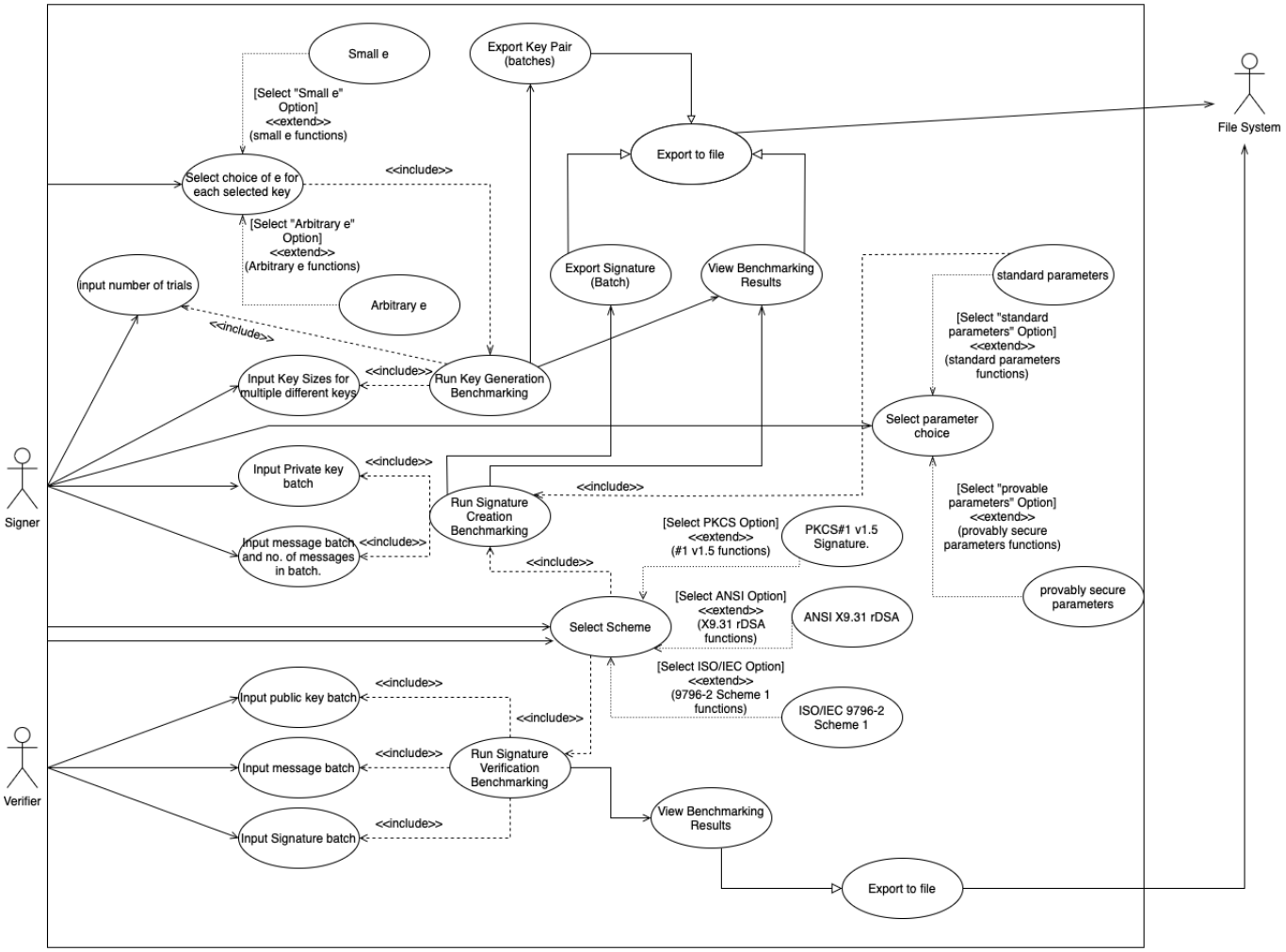


Figure 6.1: UML Use Case Diagram

Please see Appendix Appendix B.1 for a full breakdown of relevant requirements (capturing essential behaviour), and a flow-of-events type description for figure 6.1 representing completed UML use cases, derived from preceding user stories.

6.2 A Horizontal Slice: Key Generation

6.2.1 Design

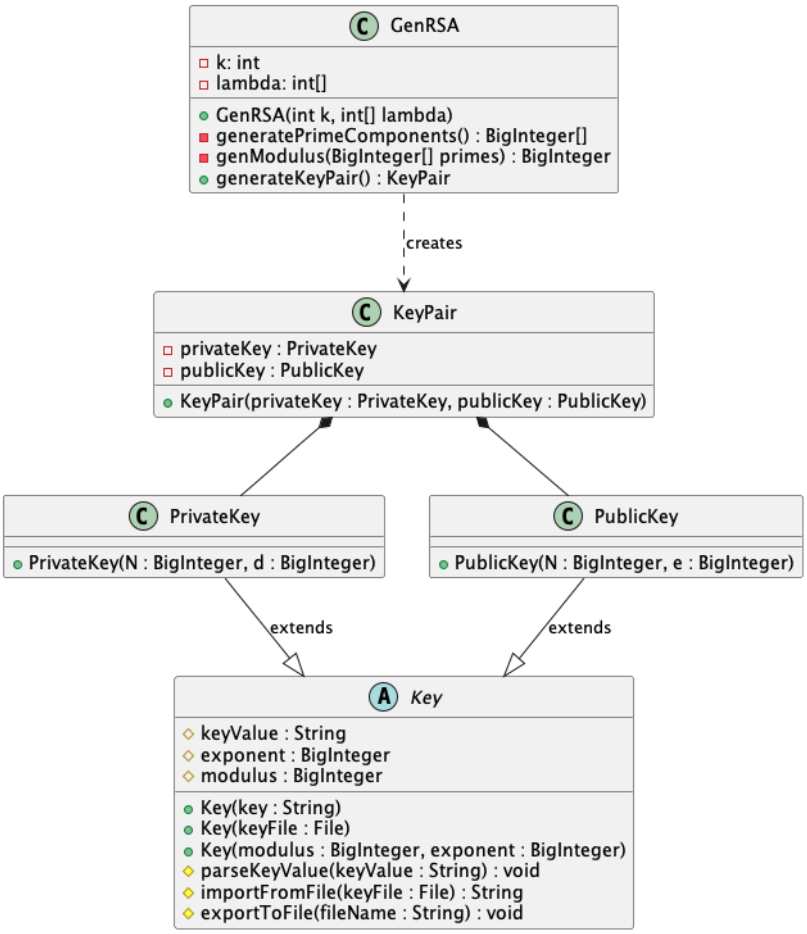


Figure 6.2: Key Generation Class Design

Figure 6.2 provides a conceptualised design view of the foundational GenRSA definition discussed in section 3.4.1 with the process broken down into smaller components corresponding to roles/entities supportive to the purpose of generating keys.

Table 6.2: GenRSA Design

Class	**Description**
KeyGenRSA	Central class responsible for generating the key pair by providing a method to fulfil the generation process.
Key	Abstract class that serves as a blueprint for the public and private keys, encapsulating the use of the modulus and an accompanying exponent with further the ability to import/-export its content.
PrivateKey / PublicKey	These classes extend from Key, specialising it for private and public key functionalities respectively.

6.2.2 Implementation

The approach taken to key generation deviates from standard practices by offering customisable options, particularly in the number of prime factors forming the modulus. Unlike traditional methods that limit the number of primes to two, the code allows the user to specify any number of primes, each with different bit sizes. The modulus and, consequently, the RSA key, are computed based on this specification. This approach aligns with the generalised key generation process outlined in Section 3.4.1. It applies as a consequence of proof implication 2 which requires the use of a non-standard number of primes. Specifically, the proofs require a 3-prime factor modulus configured as ($\lambda/4$, $\lambda/4$, $\lambda/2$). Under the 2v3PA[λ] (4.7) this is deemed to be indistinguishable from the standard two-prime factor modulus (N = pq) thus still acting in alignment with standard RSA.

```
public void initialise(int k, int[] lambda) throws IllegalArgumentException {
    if (lambda.length != k) {
      throw new IllegalArgumentException("Lambda array must have k elements."
    );
    }

    for (int bitLength : lambda) {
      keySize += bitLength;
    }

    this.k = k;
    this.lambda = lambda;
  }

```

Listing 6.1: Initialisation of parametrisable Key Generation

The suitability of Java's BigInteger class as discussed prior becomes evident from the outset. For example when the considering a foundational task: RSA Key Generation and more precisely the generation of the large primes required to encompass the modulus N. Utilising the constructor BigInteger(int bitLength, int certainty, Random rnd), it is possible to generate probable prime numbers of some arbitrary value, providing a certainty level is indicated.

```
public BigInteger[] generatePrimeComponents() {
    BigInteger[] components = new BigInteger[k];
    for (int i = 0; i < k; i++) {
      components[i] = new BigInteger(lambda[i], this.certainty, new
    SecureRandom());
    }
    return components;
  }
```

Listing 6.2: Parametrisable Prime Generation with BigInteger

This leads to the first step to be taken in any implementation of RSA: the initialisation of large prime numbers. Due to the magnitude of possible integers required in question, attempting to factorise both p and q to establish their primality with absolute certainty is computationally impractical. Instead, BigInteger employs the Miller-Rabin primality test to assess the probability of primality based on the probability

$$1 - \frac{1}{2^{certainty}}$$

For a balance between security and practicality, a certainty value between 50-100 is typical. I chose the midpoint value as a practical compromise.

```
1  private int eBitLength;
2
3  public GenRSA(int k, int[] lambda, boolean isSmallE) throws
     IllegalArgumentException {
4     initialise(k, lambda);
5     eBitLength = isSmallE ? keySize / 4 : keySize;
6   }
7
8   public BigInteger computeE(BigInteger phi) {
9     BigInteger e = new BigInteger(eBitLength, new SecureRandom());
10    while (e.compareTo(ONE) <= 0 || !phi.gcd(e).equals(ONE) || e.compareTo(
      phi) >= 0) {
11      e = new BigInteger(eBitLength, new SecureRandom());
12    }
13    return e;
14  }
```

Listing 6.3: Adaptation for specification of small e

Another novel aspect of this implementation as per implication 4 of the φ-Hiding Assumption, is the provision for a small public exponent $e \leqslant N^{1/4}$. This is initiated by providing an additional constructor that includes an isSmallE flag to specify whether a small e is desired. The bit length of e is then calculated accordingly. If a small e is chosen, its bit length is set to a quarter of the key size, ensuring compliance as per the considered proof. Otherwise, e's bit length matches that of ϕ. In either case we have it that e, belongs to the group $(Z/Z_{\phi})^{\times}$ meeting standard RSA requirements.

The remaining computation in the attempt to obtain the private exponent d is straightforward with a naturally supported modInverse operation.

```
1
2   public KeyPair generateKeyPair() {
3     BigInteger[] primes = this.generatePrimeComponents();
4     BigInteger N = genModulus(primes);
5     BigInteger phi = computePhi(primes);
6     BigInteger e = computeE(phi);
7
8     BigInteger d = e.modInverse(phi);
9     PublicKey publicKey = new PublicKey(N, e);
10    PrivateKey privateKey = new PrivateKey(N, primes, phi, e, d);
11
12    return new KeyPair(publicKey, privateKey);
13  }
```

Listing 6.4: Java Implementation of Key Generation (3.4.1)

6.3 Design

For other types of design documentation, please see Appendix B.2.

This section depicts the structure of classes to be used in the implementation of the project deliverable. For brevity, The section details the design only up to the SignatureController Assembly. The remaining components are detailed at the beginning of Appendix B.2. It can

however be noted that the application's modular structure is consistent throughout, so the design principles and patterns applied here for the signature module are similarly reflected across all other modules.

Embedded within the architecture is a deliberate emphasis on separation of concerns, following the Model-View-Controller (MVC) design pattern. In general terms, MVC separates the application into three interconnected components: the 'Model' for the application's data, the 'View' for the user interface, and the 'Controller' for managing user inputs and updating the Model and View accordingly.

6.3.1 Design of Main Controller: Orchestration of Application Flow

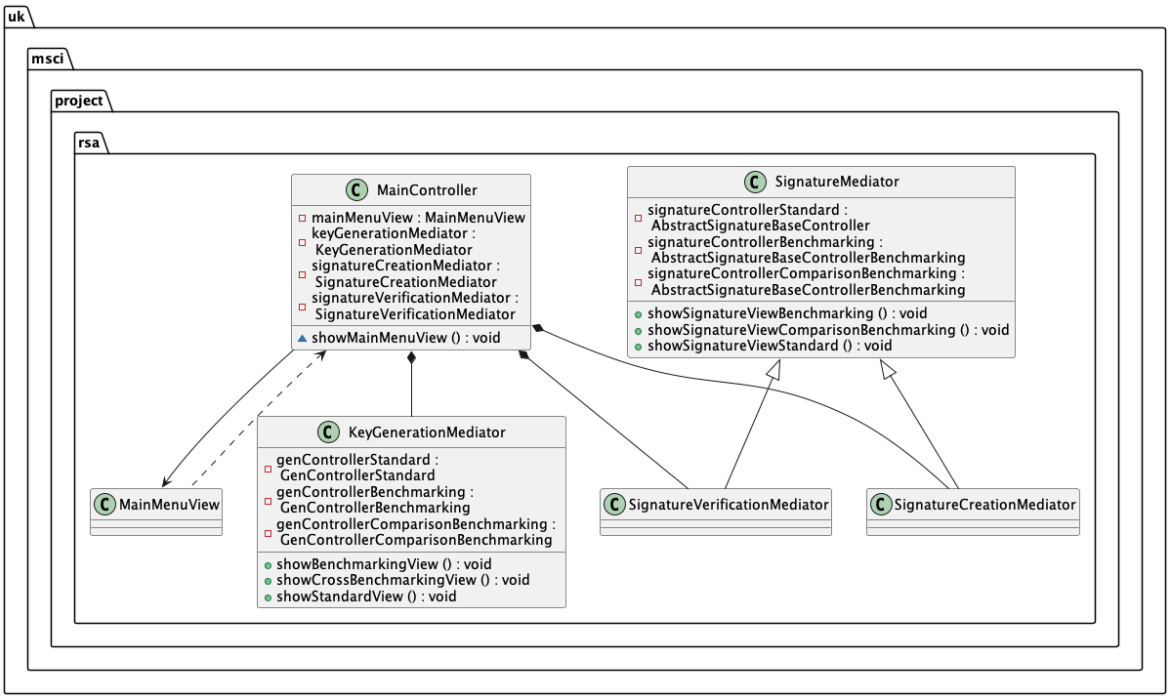


Figure 6.3: Design of Main Controller: Orchestration of Application Flow

6.3.2 MVC pattern

At the heart of the design is the MainController, which orchestrates the application flow by responding to user actions and coordinating the display of different views linked to the specialised modules.

There are two applications of the MVC coordinated by the MainController i.e., the applications two functional (pre-benchmarking) modules: the signatures and additionally the Key Generation module. Most essentially, each of these module's, controller assembly's are encapsulated using a respective mediator class. The role of the mediator classes are to act as an intermediary between the main controller of the application and controller assembly of the relevant functional module. This is required because the application is to consist of various operational modes: i.e, standard, benchmarking, and comparison benchmarking. This class ensures the appropriate controllers are utilised and configured based on the operational context and mode. The mediator pattern fulfils this by obviating the need for the MainController

to interact with each controller for an application module individually. The first of these to be considered SignatureMediator.

6.3.3 Signature Module

Design of SignatureController Assembly: Orchestration of Signature Related functionalities

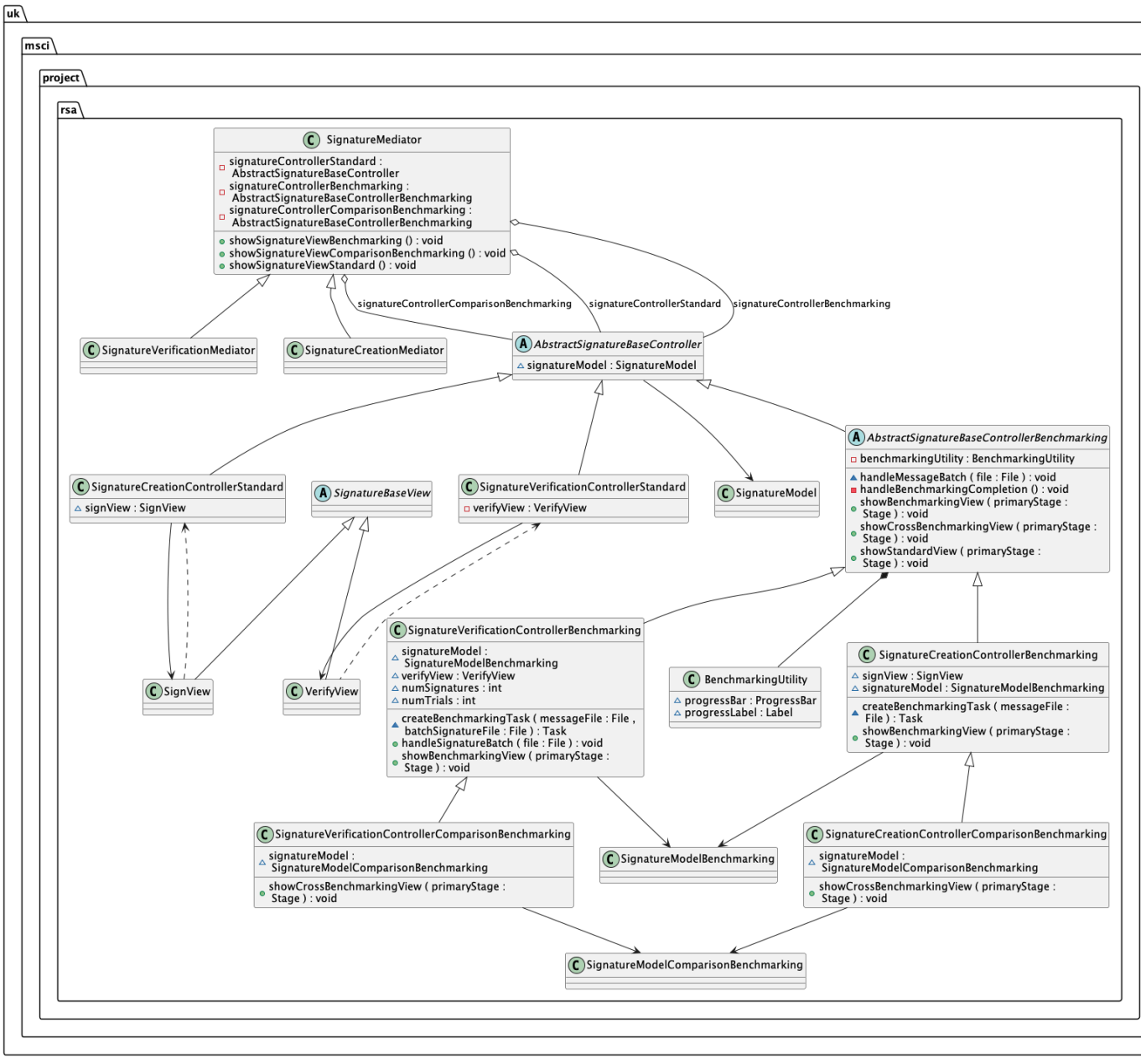


Figure 6.4: Design of SignatureController Assembly: Orchestration of Signature Related functionalities

Delving deeper into the orchestration of signature-related functionalities, the UML class diagram presents overview of the structure, facilitated by the SignatureMediator class. The SignatureMediator serves as a centralised hub acting to manage the flow between the Main-Controller and the application mode specific signature controllers. Its design encompasses concretely the subclasses: SignatureCreationMediator and the SignatureVerificationMedia-

tor, each tasked with a specific aspect of signature processing.

The SignatureCreationMediator class is responsible for the initiation and management of signature generation. It interacts with the SignatureCreationControllerStandard (which handles single use non-benchmarked signature creation processes), and with the SignatureCreationControllerBenchmarking for recording and documenting the performance of signature schemes instantiated using differing parameter types on a batch of messages. The SignatureVerificationMediator mirrors this structure for the verification processes. Both the SignatureCreationControllerBenchmarking and the SignatureVerificationControllerBenchmarking classes are equipped with a BenchmarkingUtility, which includes a ProgressBar component. This provides live feedback during benchmarking processes, enhancing the user experience by delivering a visual indication of the operation's progress from 0 to 100.

The subclass SignatureCreationControllerComparisonBenchmarking and its verification counterpart, SignatureVerificationControllerComparisonBenchmarking, extend normal benchmarking functionality to accommodate comparison benchmarking. They allow for the side-by-side evaluation of parameter types for a provided signature scheme by key size, offering insights into the relative performance of each signature scheme.

The SignatureModel and SignatureModelBenchmarking classes (see Appendix B.2) serve as a software approximation of a real-world processes of the signing and verification processes. They maintain the state of signature data and respectively handle standard or benchmarking mode operations thus fulfilling the processing of signatures and execution of benchmarking tasks if applicable.

The KeyGenerationMediator mirrors the structure of the SignatureMediator but is focused on generating cryptographic keys through the key generation model assembly which manages the GenRSA facilitated key creation.

Observer pattern

The UML diagram also hints at the observer pattern, a subset of MVC, with the Signature Controllers acting as observers to their relevant Views, which are the observables. When a user interacts with a signature view, such as initiating a signature generation request, the View notifies the SignatureController, which in turn invokes the necessary methods in both the View and the Model to fulfil the request. This decouples the user interface from the business logic, allowing for independent development and refinement of each component.

6.4 Implementation: An Extensible Signature Scheme framework

Having explored the theoretical underpinnings and practical applications of digital signature schemes, we now transition to a more technical focus of how signature schemes can be concretely implemented with a generalised framework.

The foundation of this is provided by SigSchemeInterface, an interface that outlines the essential functionalities required for any digital signature scheme. This interface serves as a contract for implementing classes, ensuring that fundamental operations such as signing, verifying, and data conversion between different formats are standardised and consistent.

```
1 public interface SigSchemeInterface {
2
```

```
3   byte[] sign(byte[] M) throws SignatureException, DataFormatException;
4
5   boolean verify(byte[] data, byte[] signature) throws SignatureException,
      DataFormatException;
6
7   BigInteger OS2IP(byte[] EM);
8
9   byte[] I2OSP(BigInteger m) throws IllegalArgumentException;
10
11  BigInteger RSASP1(BigInteger m);
12
13  BigInteger RSAVP1(BigInteger s);
14 }
15
```

Listing 6.5: SigSchemeInterface Interface code

Signing a message m (sign(m)) works as follows:

1. Encode the message: encodeMessage(m).
2. Convert the message to representative integer form: OS2IP(m).
3. Using a private key sk, calculate the signature representative s using RSA: RSASP1(sk, m).
4. Convert the signature (integer) representative back to a byte string: I2OSP(s).

Verification is the reverse, adapted accordingly. Given a signature s (verify(s)):

1. Convert s into an integer representative: OS2IP(s).
2. Using public key vk, calculate the message representative m: RSAVP1(vk, s)
3. Convert m to a byte string: I2OSP(m).
4. Verify the byte string using verifyMessage.

This generalised flow presents two further methods: 'encodeMessage', an algorithm to format data before performing a cryptographic signing operation on its integer representation, and 'verifyMessage', an algorithm that verifies the signature after a cryptographic verification operation has been performed on its integer representation.

Building upon the SigSchemeInterface, the abstract class SigScheme is introduced as a concrete implementation of the Interface for implementing RSA-based signature schemes. This class provides standardised implementations of the interface's methods and includes additional functionality specific to RSA operations. It encapsulates key components like the RSA modulus, exponent, and message digest, contributing to a modular and extensible design suitable for various RSA-based signature schemes.

```
1
2   /**
3    * Converts a BigInteger to an octet string of length emLen is the byte
       length of the RSA modulus.
4    */
```

```
  public byte[] I2OSP(BigInteger m) throws IllegalArgumentException {
    return ByteArrayConverter.toFixedLengthByteArray(m, this.emLen);
  }

  /**
   * Calculates the RSA signature of a given message representative by raising it to the power of
   * the private exponent as outlined by the RSA algorithm.
   */
  public BigInteger RSASP1(BigInteger m) {
    BigInteger s = m.modPow(this.exponent, this.modulus);
    return s;
  }

  /**
   * Facilitates the verification of RSA signature by raising it to the power of the public exponent
   * as outlined by the RSA algorithm.
   */
  public BigInteger RSAVP1(BigInteger s) {
    return this.RSASP1(s);
  }

}

```

Listing 6.6: Implementation of RSA primitives

We note that the implementation of RSAVP1 re-uses RSASP1. This design choice is viable because the exponent field in the SigScheme class is generic and can represent either the public or private exponent, depending on the type of key the scheme is instantiated with. This reuse simplifies the code and centralises the modular exponentiation logic in one place.

```
  public BigInteger OS2IP(byte[] EM) {
    return new BigInteger(1, EM);
  }

```

For octet string to integer conversion, the BigInteger(int signum, byte[] magnitude) constructor provides a concise application of the concept succinctly converting the byte array that represents an octet string. The signum is set to 1 (since messages are non-negative) and the byte array of the message acts as the magnitude. The end result is a BigInteger representation of the message, which can then be used in RSA signing or verification processes.

```
  // Abstract method to be implemented by derived classes for encoding
  protected abstract byte[] encodeMessage(byte[] M) throws DataFormatException;

  public boolean verifyMessage(byte[] M, byte[] S)
      throws DataFormatException {
    BigInteger s = OS2IP(S);
    BigInteger m = RSAVP1(s);
    byte[] EM;
    try {
      EM = I2OSP(m);
```

```
    } catch (IllegalArgumentException e) {
      return false;
    }

    byte[] EMprime = encodeMessage(M);

    return Arrays.equals(EM, EMprime);
  }

```

Listing 6.7: Signature Scheme specialisation methods

The SigScheme class provides two further methods as compared to the interface with encodeMessage and verifyMessage. The encodeMessage method is left abstract because the encoding of messages is essentially what distinguishes different signature schemes. On the other hand, verifyMessage has a standardised implementation because, in many cases, it simply reverses the signing process. Since the core logic of this reversal often follows a standard procedure that is largely independent of the specifics of the message encoding, it is feasible to provide a standardised implementation for verifyMessage within the SigScheme class.

Message Recovery

Message recovery schemes, such as ISO/IEC 9796-2 Scheme 1, represent a special case, thereby challenging the notion of standardised implementations for key methods like sign and verifyMessage.

```
public abstract class SigScheme implements SigSchemeInterface {

  byte[] nonRecoverableM;

  byte[] recoverableM;
}
public class ISO_IEC_9796_2_SCHEME_1 extends SigScheme {
  @Override
  public byte[] sign(byte[] M) throws DataFormatException {
    byte[] S = super.sign(M);
    // Extract m2 from the original message M using the computed m2's length
    if (m2Len > 0) {
      nonRecoverableM = Arrays.copyOfRange(M, m1Len - m2Len, m1Len);
    }
    return S;
  }
}

```

Listing 6.8: Implementation changes for Message Recovery Schemes

The sign method for these schemes needs to be able to set the non recoverable message field of the class, to allow the respective portion message to be returned to the user, as part of the conclusion to the signature creation process. Hence, the ISO/IEC 9796-2 Scheme 1 class includes an overridden implementation of sign modified accordingly to account for this.

Along these same lines the verifyMessage method which we omit for brevity has to incorporate logic related to the separation of the message into recoverable/non-recoverable parts into the

verification process. During verification, the verifyMessage method must correctly reconstruct the original message from these two parts.

6.5 Implementation: Enabling a Provably Secure hash output

This implementation includes support for an unusually large hash output ($|H()| \geq \frac{\lambda}{2}$) as required by both proofs. This is accomplished using variable length hash functions like SHAKE-128, or via MGF1 applied to fixed-length functions such as SHA-256.

```
1  public class DigestFactory {
2    public static MessageDigest getMessageDigest(DigestType digestType)
3        throws NoSuchAlgorithmException, NoSuchProviderException {
4      switch (digestType) {
5        case SHA_256 -> {return MessageDigest.getInstance("SHA-256");}
6        case MGF_1_SHA_256 -> {return MessageDigest.getInstance("SHA-256");}
7        case SHA_512 -> {return MessageDigest.getInstance("SHA-512");}
8        case MGF_1_SHA_512 -> {return MessageDigest.getInstance("SHA-512");}
9        case SHAKE_128 -> {Security.addProvider(new BouncyCastleProvider());
10         return MessageDigest.getInstance("SHAKE128", BouncyCastleProvider.
       PROVIDER_NAME);
11       }
12       case SHAKE_256 -> {Security.addProvider(new BouncyCastleProvider());
13       return MessageDigest.getInstance("SHAKE256", BouncyCastleProvider.
       PROVIDER_NAME);}
14       default -> throw new NoSuchAlgorithmException("Unsupported digest type:
        " + digestType);
15     }
16   }
17 }
18
```

Listing 6.9: Retrieval of Message Digest Instances

An enumeration (enum) 'DigestType' classifies each hash function. To abstract their creation, a factory pattern is used, which provides a consistent interface to obtain 'MessageDigest' objects for each signature scheme. Notably, for SHA-3 family hashes like SHAKE-128 and SHAKE-256, BouncyCastle cryptographic API is used due to the absence of native Java support.

```
1  public abstract class SigScheme implements SigSchemeInterface {
2
3    boolean isProvablySecureParams;
4    int hashSize;
5
6    public SigScheme(Key key, boolean isProvablySecureParams) {
7      initialise(key);
8      this.isProvablySecureParams = isProvablySecureParams;
9    }
10
11 }
```

Listing 6.10: Adaptation of Signature Scheme initialisation for specification of provably secure hash output

To facilitate scheme instantiation with a larger hash output, constructors include an 'isProvablySecureParams' flag. This flag aids in specifying whether an enhanced security hash output is required, allowing the bit length of the hash size to be computed accordingly.

```
public void setDigest(DigestType digestType, int customHashSize)
      throws NoSuchAlgorithmException, InvalidDigestException,
    NoSuchProviderException {
    md.reset();
    this.currentHashType = digestType;
    this.md = DigestFactory.getMessageDigest(digestType);
    this.hashID = getHashID(currentHashType);

    if (digestType == DigestType.SHA_256 || digestType == DigestType.SHA_512)
     {
      this.hashSize = md.getDigestLength(); // Fixed size
    } else if (isProvablySecureParams) {
      this.hashSize = (emLen + 1) / 2; // Provably secure size
    } else {
      if (customHashSize <= 0) {
        throw new IllegalArgumentException("Custom hash size must be a
    positive integer");
      }
      this.hashSize = customHashSize; // Custom size for variable-length hash
     types
    }
  }

```

Listing 6.11: Configuration of Hash Size

The setDigest method configures the hash size based on the hash type and operational parameters. For fixed size hashes (e.g., SHA-256, SHA-512), it sets the size corresponding to the digest type. For variable-length hashes (e.g., SHAKE-128, SHAKE-256), it offers two options: a preset provably secure size (half the length of the modulus) or a user-defined custom size.

After the hash function is determined and the hash output size configured, marking the final step of the hashing process, the computation of the actual hash within the signature scheme is required.

```
public byte[] computeHashWithOptionalMasking(byte[] message) {
    if (!(currentHashType == DigestType.SHA_256 || currentHashType ==
    DigestType.SHA_512)) {
      if (currentHashType == DigestType.MGF_1_SHA_256
          || currentHashType == DigestType.MGF_1_SHA_512) {
        return new MGF1(this.md).generateMask(message, this.hashSize);
      } else {
        return computeShakeHash(message);
      }
    } else {
      return computeHash(message);
```

```
11      }
12    }
13
```

Listing 6.12: Final Computation of hash

Finally, the computeHashWithOptionalMasking method calculates the hash within the signature scheme. It handles variable length hash functions distinctively, especially for MGF1, which requires generating a mask using the determined hash size.

6.6 Implementation: Application Overview

The application comprises four main portals, each linked to functionalities essential for facilitating digital signatures. These portals allow transition between standard and benchmarking-specific operations.

1. **Main Menu:** The gateway to key and signature functionalities within the application.
2. **Key Generation:** Allows generation of RSA key pairs, either singularly for standard operations or in batches for benchmarking purposes.
3. **Signature Creation:** Facilitates creation of individual signatures in standard mode or signature batches in benchmarking mode for deterministic signature types.
4. **Signature Verification:** Enables validation of individual or batch signatures against messages or message batches, respectively, in standard or benchmarking modes.

Focus on Comparison Benchmarking:

The application offers four modalities, categorised into standard mode and a suite of benchmarking modes. This section focusses on the preset Comparison Benchmarking mode since it most directly aligns with the project's aim to assess the burden on computational time that arises when using provably secure parameters in deterministic RSA signature schemes.

Comparison Benchmarking mode offers a direct comparative analysis between standard and provably secure key configuration groups (with user selected hash functions for each group). This allows evaluation of the performance of the Standard and Provably Secure groupings, with all trials repeated for each hash function and key configuration within each group. Results are then presented side by side in a table and/or overlaid graphs for each entered key size.

6.6.1 Main Menu

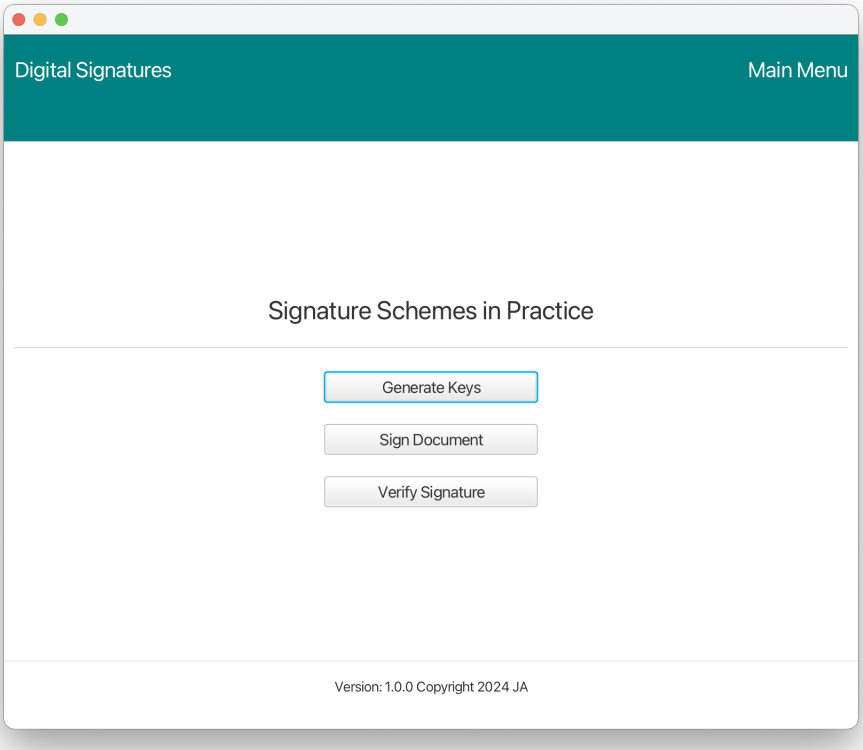


Figure 6.5: Main Menu

The main menu screen of the application presents a straightforward and functional layout, serving as the gateway to its core features via menu buttons. Its design is realised through an FXML layout file, which defines the graphical user interface elements such as buttons. Each button on the main menu is associated with a specific functionality and is observable by the virtue of its MainMenu domain object class. On launch, the MainController registers observers on the observable components of the main menu screen. User interactions prompt the corresponding observer to trigger its effects. The equivalent pattern is followed for all other components in the design of the program. From this point we assume its usage.

6.6.2 Key Generation

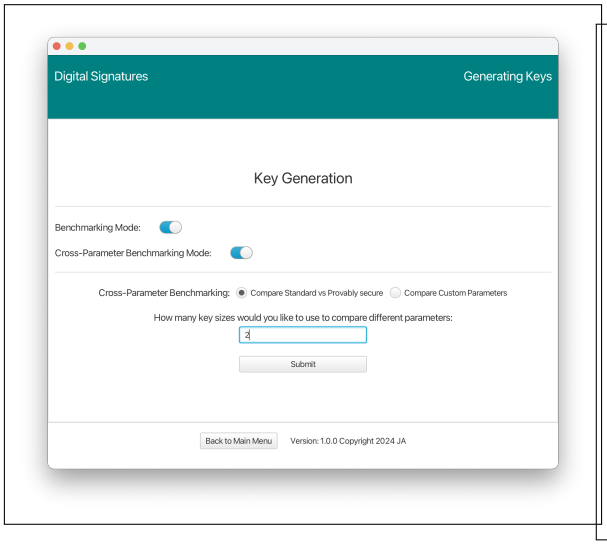


Figure 6.6: Key Generation (number of key sizes)

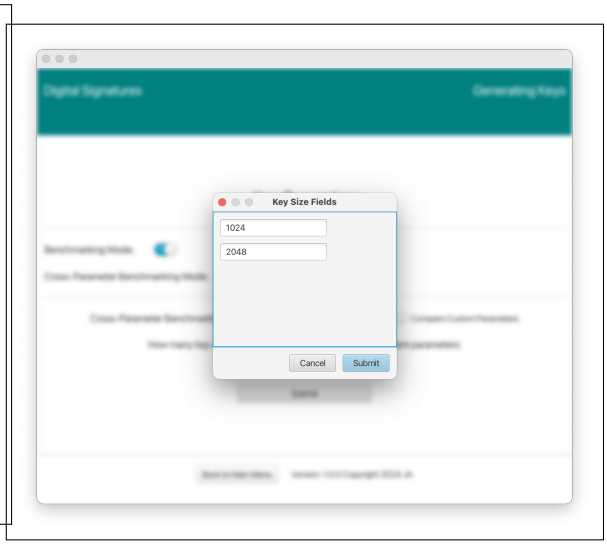


Figure 6.7: Key Generation (key sizes)

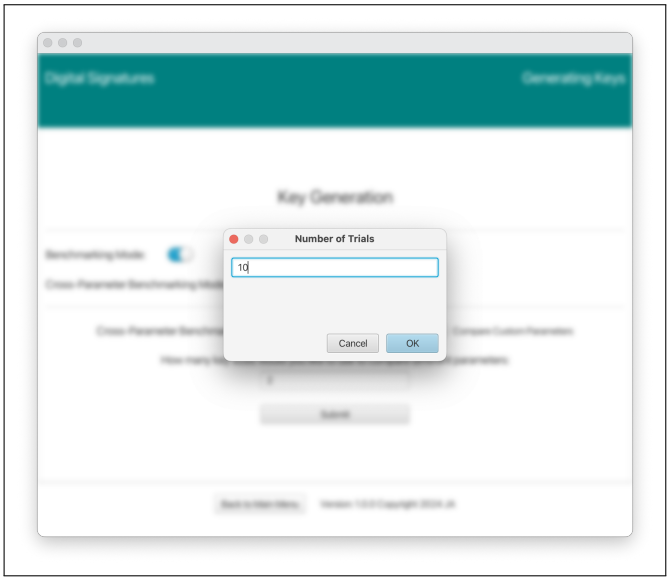


Figure 6.8: Key Generation (number of trials)

The key generation interface for comparison benchmarking is activated through a toggle switch. This interface offers a radio button to select the default comparison benchmarking mode, which compares standard and provably secure key configurations. Users specify the number of key sizes they wish to test, and corresponding input fields are displayed for each size (e.g., Key Size 1: 1024, Key Size 2: 2048).

For each entered key size, the application generates two groups of key configurations:

- A **Standard Parameters** group, with both 2-prime and 3-prime modulus configurations and arbitrary e values.
- A **Provably Secure Parameters** group, with 2-prime and 3-prime modulus configurations and small e values.

Users then input the desired number of trials for key generation benchmarking, such as 10 trials. The application executes these trials for each of the four key configurations at the selected key sizes.

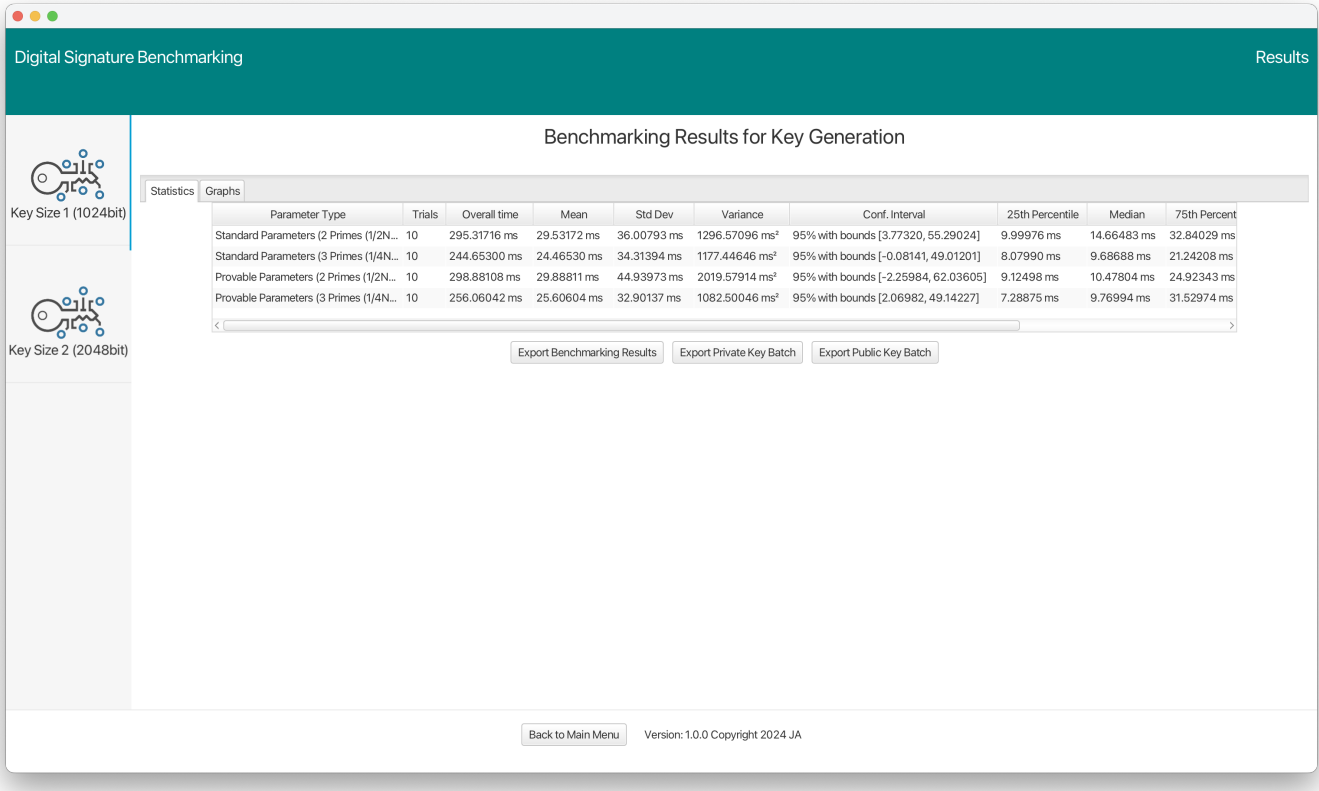


Figure 6.9: Key Generation (Results Screen)

Upon completion of benchmarking, the ResultsController is engaged to display the results from benchmarking. The results screen contains a side pane with buttons corresponding to each individual key size previously entered. Within each side tab, the results table displays a row by row sequence of statistical metrics for all 4 key configurations. The ordering is by parameter set (i.e., 2 key configurations for standard parameter key results first, followed by the 2 key configurations for provably secure parameter results second). Below the results table, the application also provides the functionality to export the benchmarking results and if desired public or private key batches corresponding to a global set of keys matching the key configurations enumerated in a sequence for all key sizes.

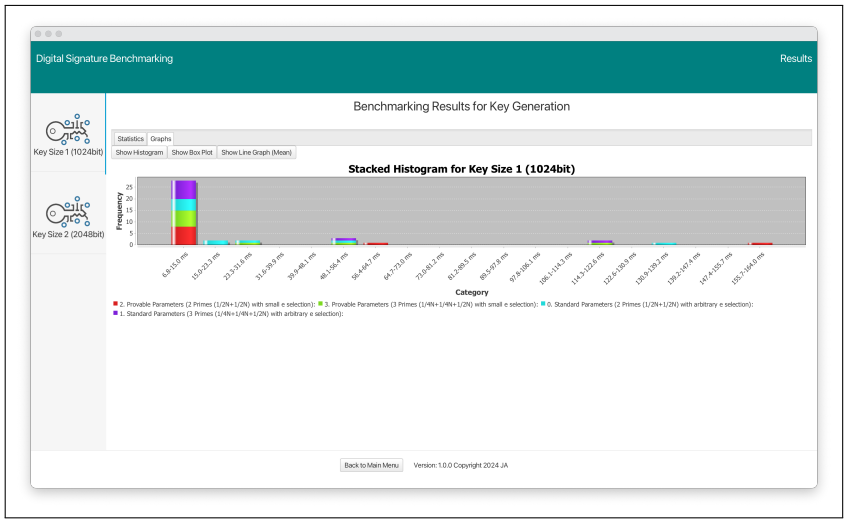


Figure 6.10: Key Generation: Overlaid Histogram

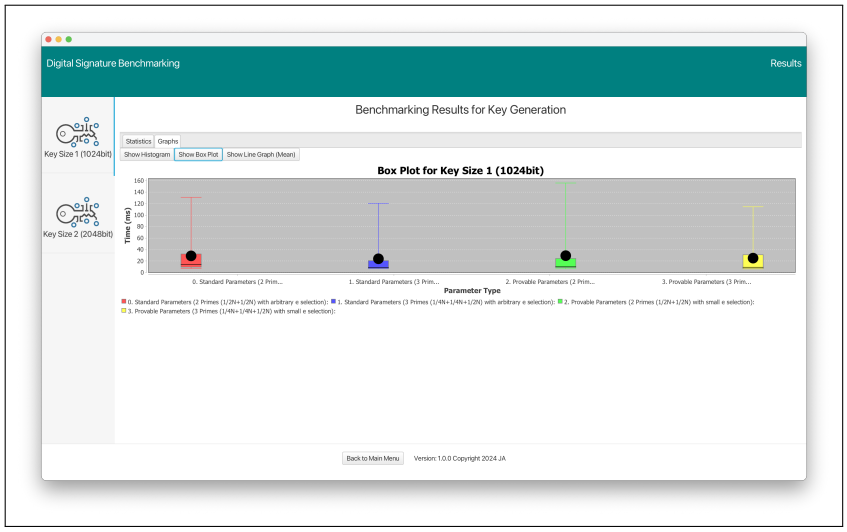


Figure 6.11: Key Generation: Overlaid Box plot graph

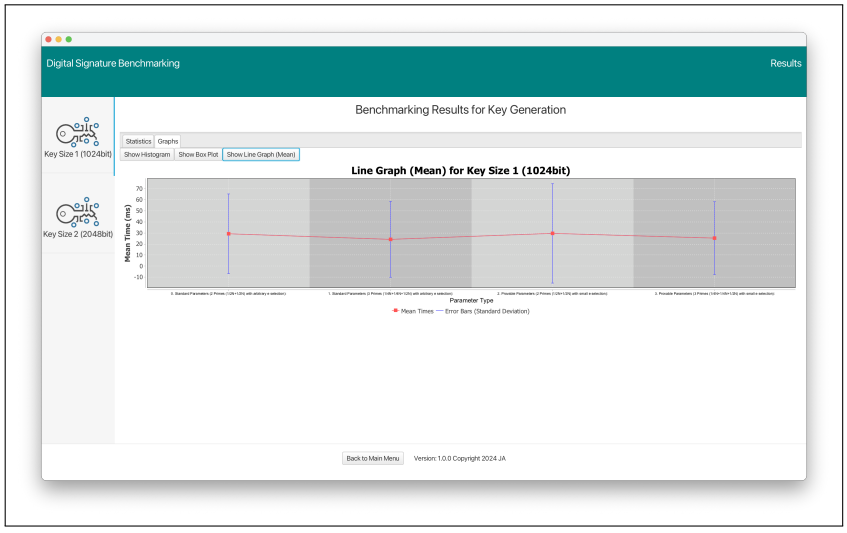


Figure 6.12: Key Generation: Overlaid Line graph for mean times

An alternative graphical representation of the results can be accessed by navigating to the "Graphs" tab within the central tab pane, where the results table is situated. This tab offers support for three distinct types of graphs, each containing data from all key configurations

mapped from the table for easy comparison. These graph types include a stacked histogram, which visualises the distribution of results among configurations for each key size. Additionally, box plots are provided for inferring the distribution of data among configurations. Lastly, mean time line graphs (with error bars for standard deviation) facilitate comparison of average performance and variability among configurations.

6.6.3 Signature Generation

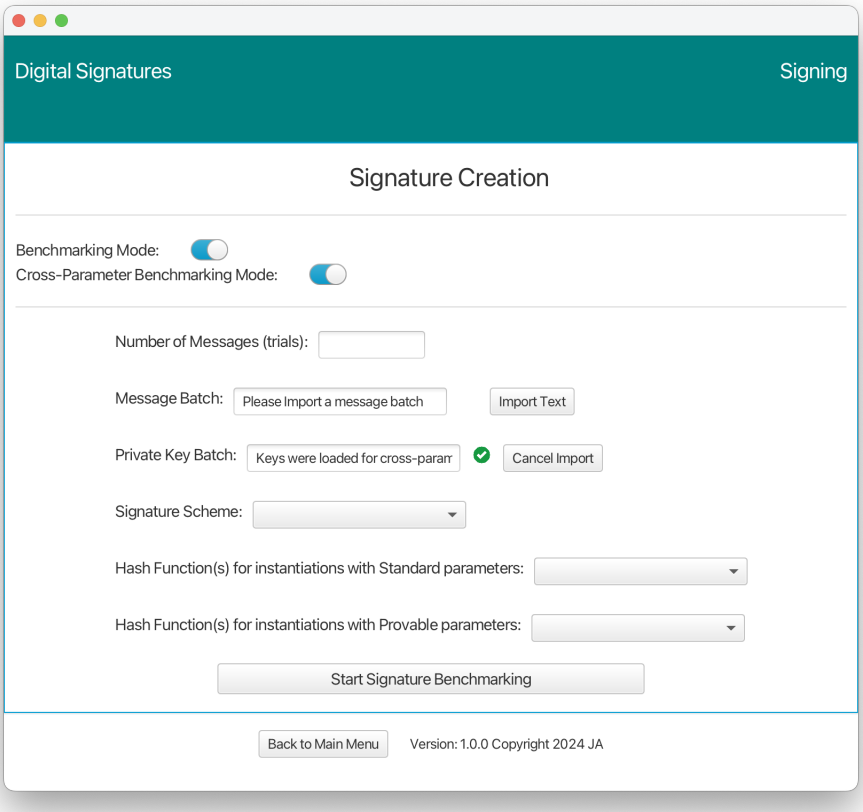


Figure 6.13: Signature Generation (Comparison Benchmarking)

The initial signature generation screen for comparison benchmarking is activated by completing a key generation benchmarking run in comparison mode. On completion of this, the private key batch (which is also available for export from the results table tab) is preloaded into the signature creation controller. Thus when signature generation option is selected from the main menu, the comparison benchmarking screen is displayed rather than the default benchmarking screen. This explains the state of cross-parameter benchmarking toggle being set as switched on, as well as, specialised options appearing such as the application displaying a visual indication that a comparison-compatible private key batch was preloaded.

```
Message 0
Message 1
Message 2
Message 3
Message 4
Message 5
Message 6
Message 7
Message 8
Message 9
```

Figure 6.14: Signature Creation (Test message batch (testMessages.txt))

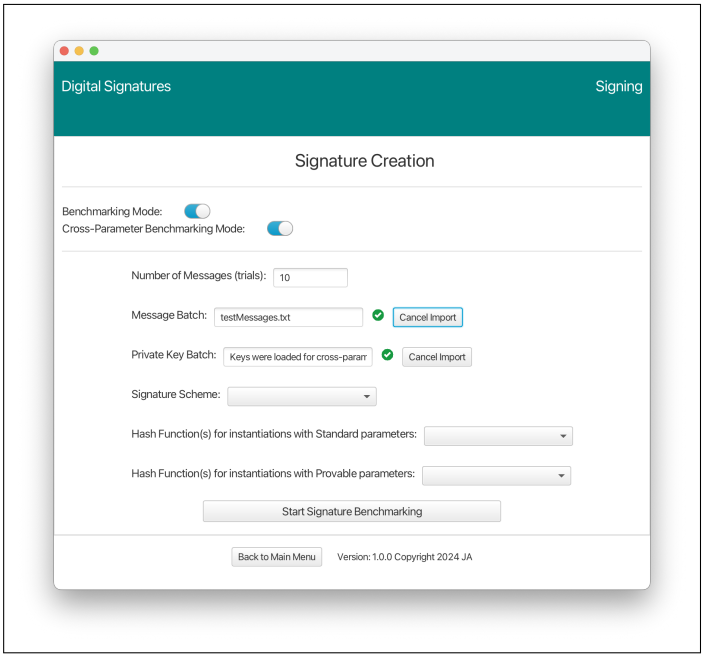


Figure 6.15: Signature Creation (Imported message batch)

The signature creation screen is first organised with a pair of fields related to the importing of a message batch. In the backdrop, the SignatureController class waits to be notified of interaction with the message batch import button. The SignatureController expects the message batch as a non interrupted and new line separated sequence of messages. Upon attempt at an import of a message batch, the SignatureController validates that the number of messages in the message batch candidate, matches the number entered in the adjacently above field, and prevents the import in case of a mismatch. On successful validation it updates the screen with a checkmark indicator that provides visual confirmation that the message batch was successfully imported. The interface also includes a dropdown menu labeled signature scheme giving users the option to select the desired signature scheme to be applied.

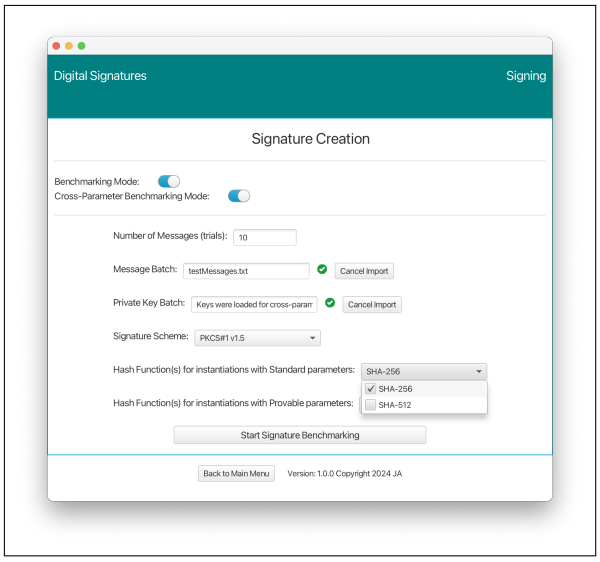


Figure 6.16: Signature Creation (Standard Parameter Hash Functions)

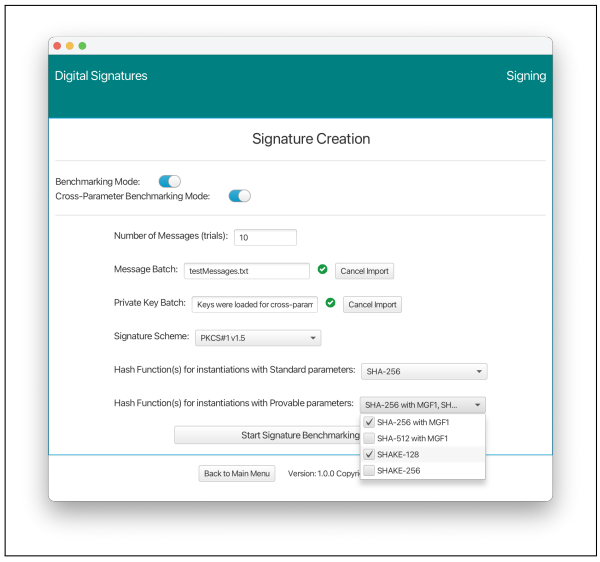


Figure 6.17: Signature Creation (Provably Secure Parameter Hash Functions)

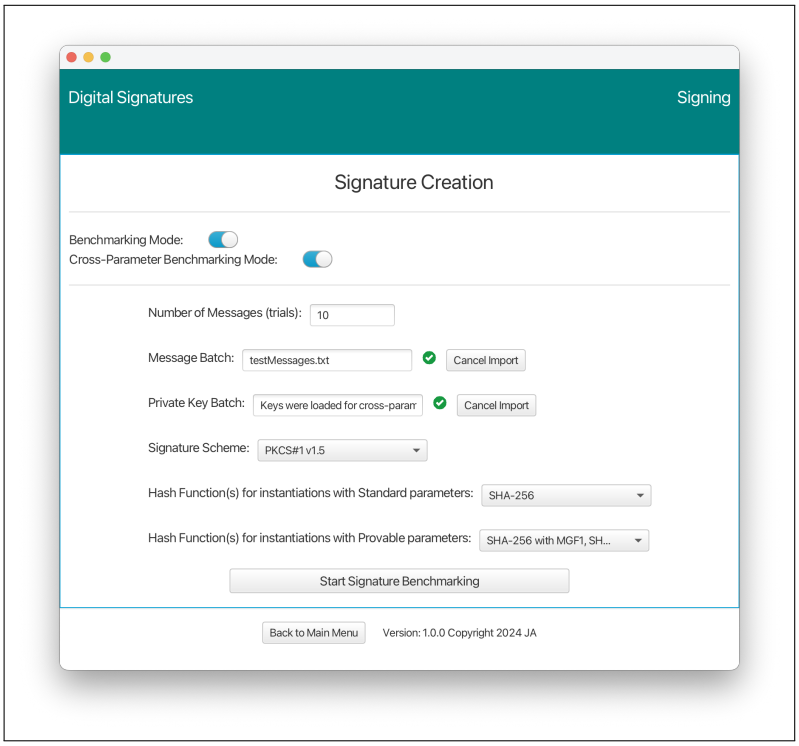


Figure 6.18: Signature Creation (All inputs provided)

In comparison benchmarking mode for signature generation, users select hash functions for standard (fixed-length) and provably secure (variable-length) parameters via a multi-choice drop-down menu. The process involves:

Computing signatures for each selected fixed-length hash function, using the first two key configurations (standard parameters). Computing signatures for each chosen variable-length hash function, using the last two key configurations (provably secure parameters).This is done sequentially for each key size.

The "Start Signature Benchmarking" button initiates a series of checks and processes. The SignatureController validates the message batch, private key batch, signature scheme, and

hash function choices. If any issues are detected, an alert prompts the user to correct them.

Upon passing these checks, the SignatureModel executes the benchmarking task, creating a batch of signatures for the message batch using the designated signature scheme and hash function-key combinations. A progress bar indicates progress in real time.

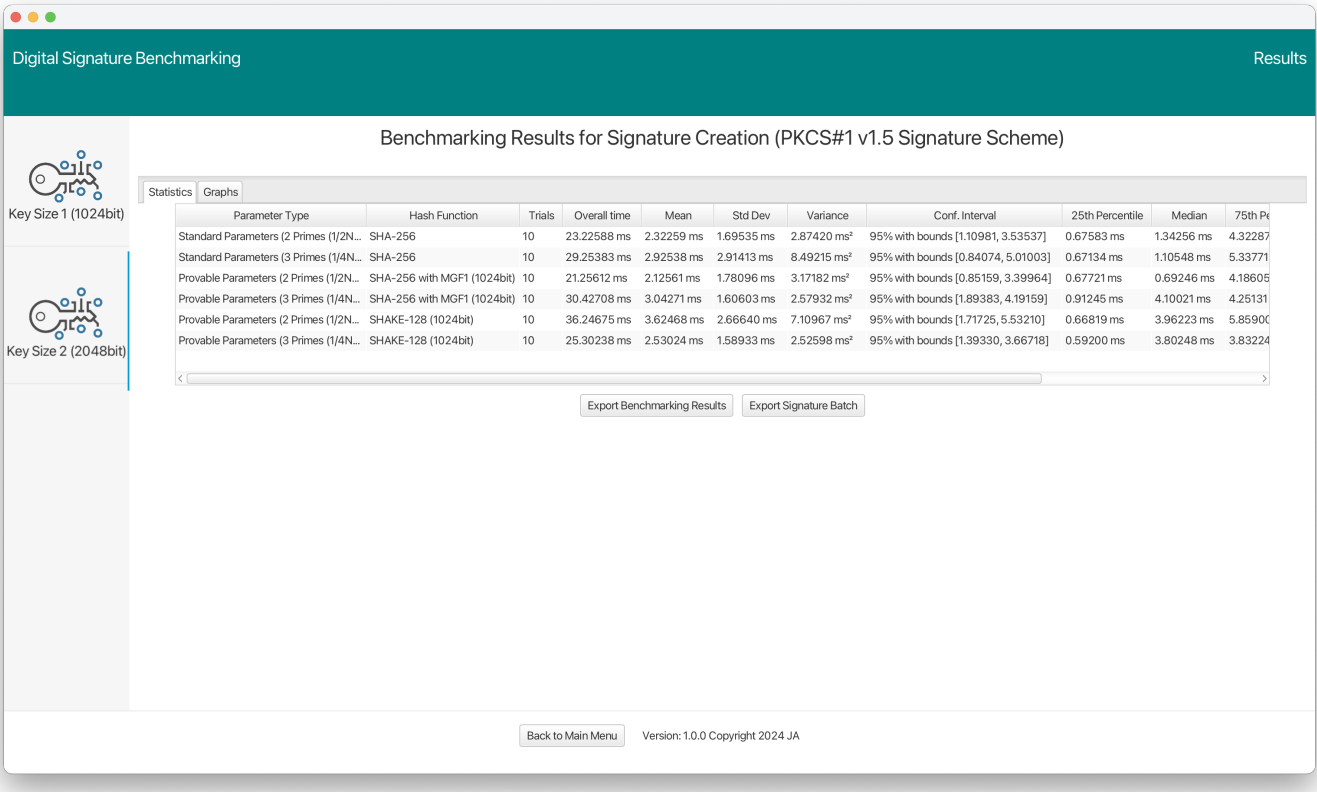


Figure 6.19: Signature Generation (Results Screen)

After completing signature generation benchmarking, the ResultsController displays the results. The results screen features a side pane with tabs for each key size entered. Each tab reveals a results table displaying statistical metrics for all four key configurations.

The table format aligns with the user's hash function choices. For example, selecting one hash function (e.g., SHA-256) for standard parameters results in two rows for this set. For provably secure parameters, if two functions like SHA-256 with MGF1 and SHAKE-128 are chosen, they each generate two rows, totalling four rows. This format leads to six rows per key size, each row representing signature computations for the full message batch.

Below the table, options are available to export benchmarking results and signature batches for each key size. The exported signature batch aligns with the key configurations and hash function combinations used, covering all key sizes in a single file.

Signature Generation Results Screen Graphs

Figure 6.20: Signature Generation: Overlaid Histogram

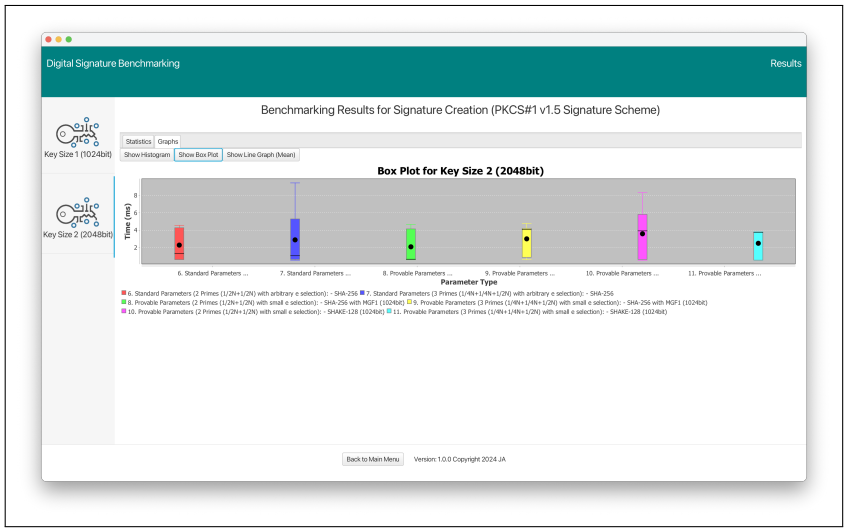


Figure 6.21: Signature Generation: Overlaid Box plot graph

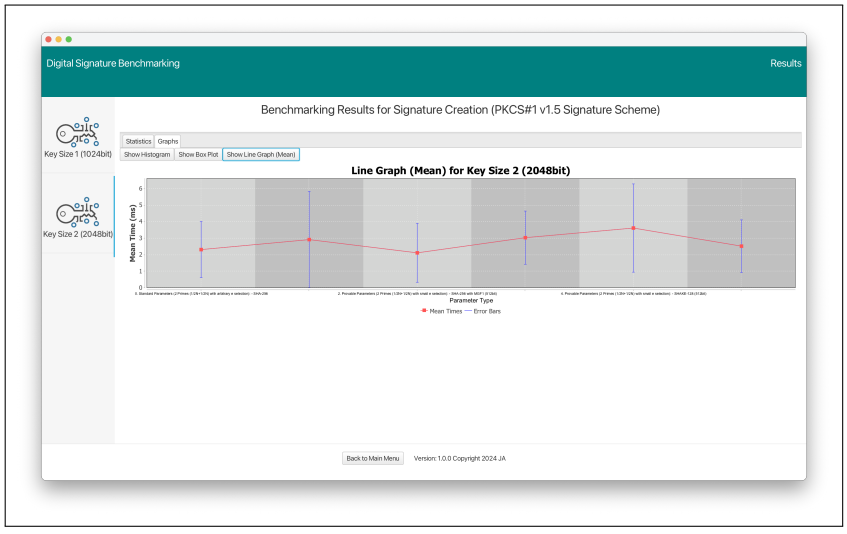


Figure 6.22: Signature Generation: Overlaid Line graph for mean times

Signature generation results can also be visualised graphically, accessible under the "Graphs" tab in the central pane. These graphs, correlating with key configurations from the results table, offer an alternate method for interpreting data.

6.6.4 Signature Verification

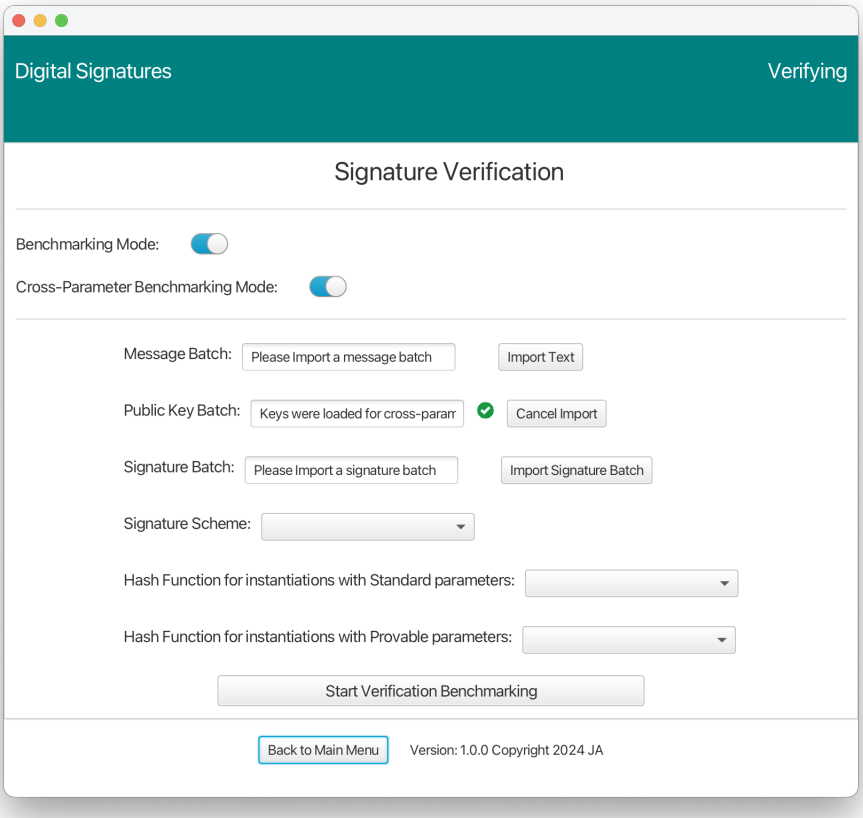


Figure 6.23: Signature Verification (Comparison Benchmarking)

Similar to the process for signature generation, the signature verification screen for comparison benchmarking is accessible after completing a key generation benchmark in comparison mode. This procedure also includes preloading a batch of keys. However, the key distinction lies in the type of key batch loaded. In the case of signature verification, it's a public key batch that gets preloaded, as opposed to a private key batch used for signature generation.

```
Message 0
Message 1
Message 2
Message 3
Message 4
Message 5
Message 6
Message 7
Message 8
Message 9
```

Figure 6.24: Signature Verification (Test message batch (testMessages.txt))

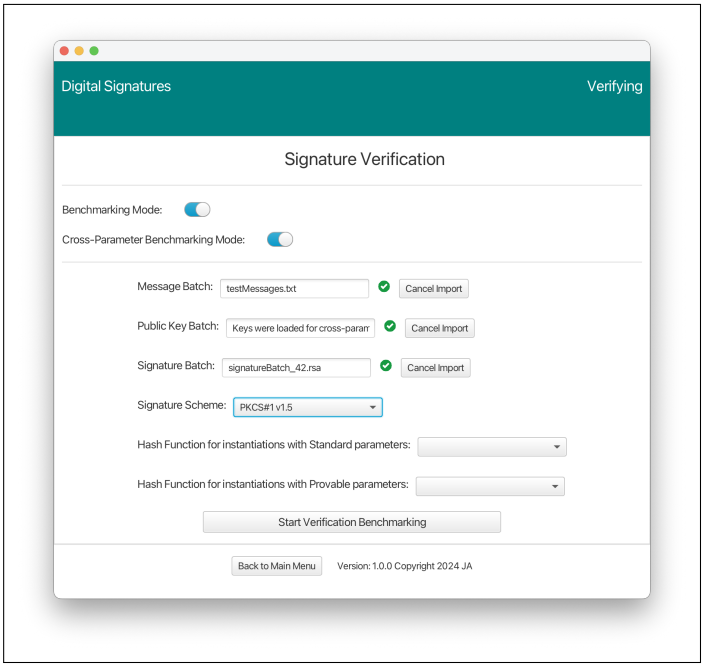


Figure 6.25: Signature Verification (Imported message batch)

Aside from the preloaded public key batch used for comparison benchmarking, the signature verification screen includes fields for importing both message and signature batches. For correctness, it's necessary that the submitted message batch aligns with the submitted signature batch. Moreover, these batches must be compatible with the public key batch, which, in turn, is associated with the private key batch employed during the prior signature generation phase.

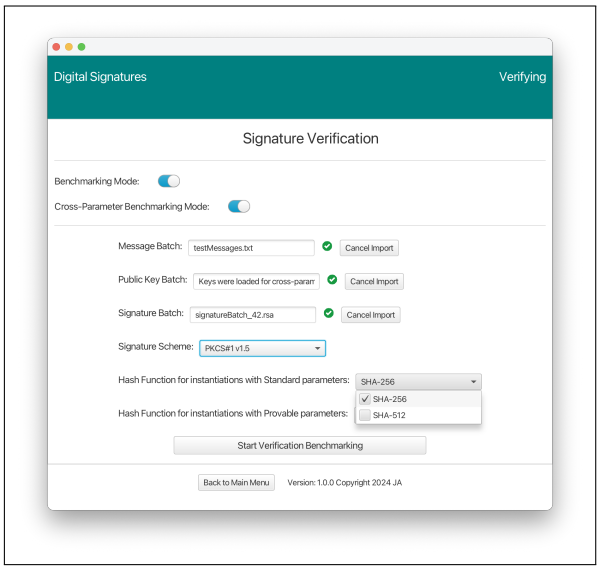


Figure 6.26: Signature Verification (Standard Parameter Hash Functions)

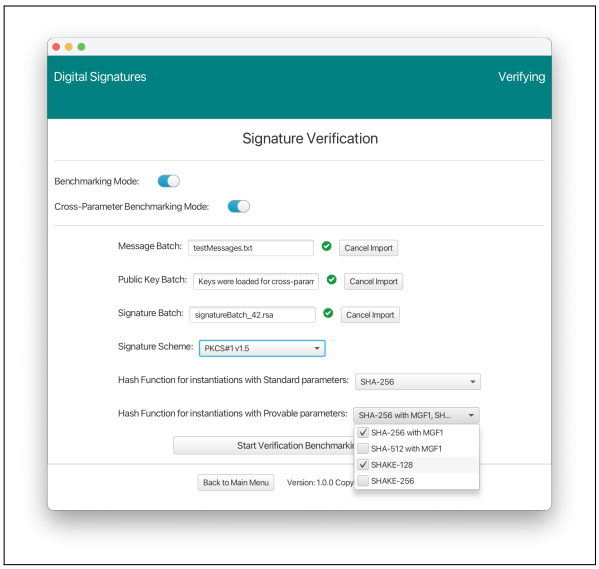


Figure 6.27: Signature Verification (Provably Secure Parameter Hash Functions)

In signature verification (comparison) benchmarking, hash function selection must align with those used in signature generation to ensure correctness. The process starts with the user choosing matching hash functions for each parameter type. This is vital because the hash functions used must correspond to the message-signature pairs created in the signature gen-

eration phase, which used the private key batch related to the currently imported public key batch.

Upon selecting "Start Verification Benchmarking," the SignatureController validates the number of signatures against the message batch and chosen hash functions. If there's a mismatch, an alert notifies the user of the discrepancy. Following successful validation, the SignatureModel takes over to execute the verification benchmarking. This involves comparing the signature batch with the message batch and public keys, using the selected signature scheme and hash functions. The process generates batches of verification results for each entered key size, and a progress bar provides real-time progress updates.

Signature Verification Results Screen

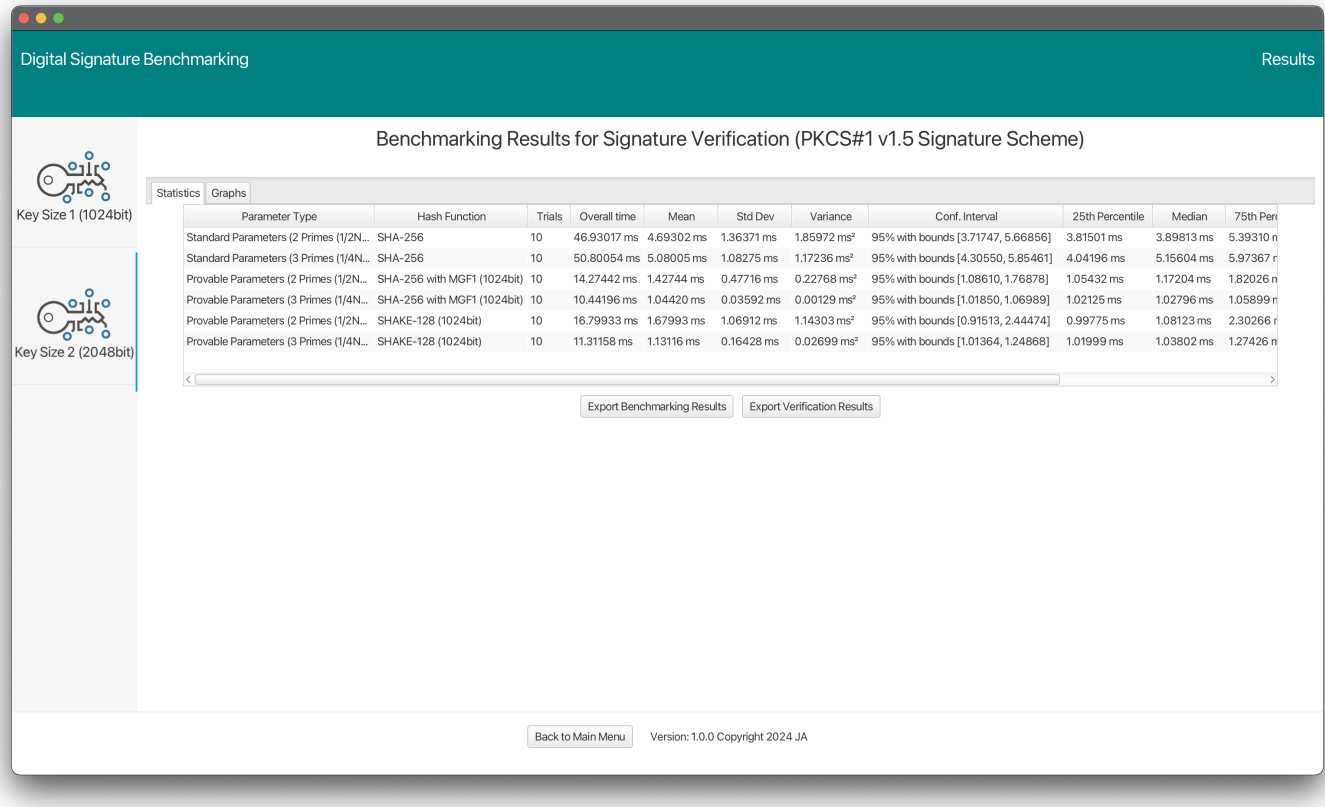


Figure 6.28: Signature Verification (Results Screen)

Once the signature verification benchmarking concludes, the ResultsController takes over to showcase the obtained results. The layout of this results screen is largely as was shown for signature generation. The distinction is the enabling of the export of verification results for each key size. displayed below each results table.

Figure 6.29: Signature Verification: Overlaid Histogram

Figure 6.30: Signature Verification: Overlaid Box plot graph

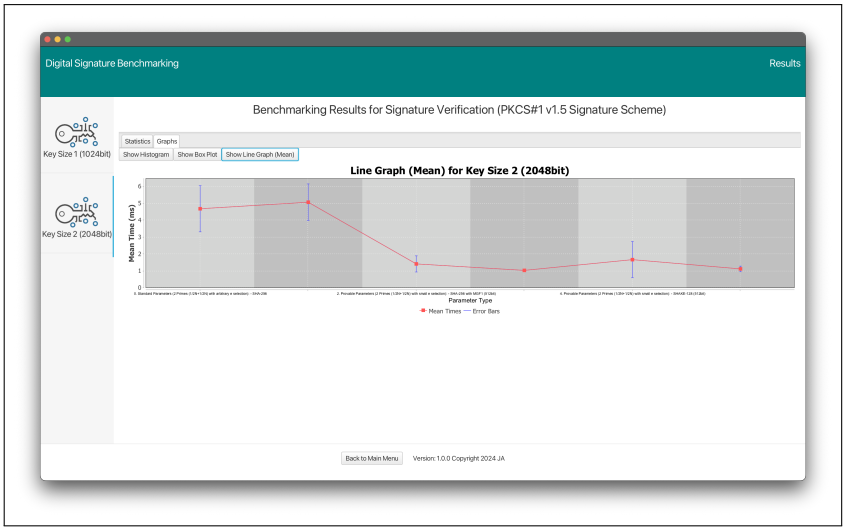


Figure 6.31: Signature Verification: Overlaid Line graph for mean times

6.6.5 Signature Processing for Message Recovery Schemes

In the Signature processes, when utilising the message recovery ISO/IEC 9796-2 Scheme, there is a specific approach for handling non-recoverable portion when signature generation is performed and recovered portion of messages when signature verification is performed.

```
Message 0
Message 1
Message 2
Message 3
Message 4
Message 5
Message 6
Message 7
Message 8
Message 9
```

Figure 6.32: Signature Creation (Test message batch (testMessages.txt))

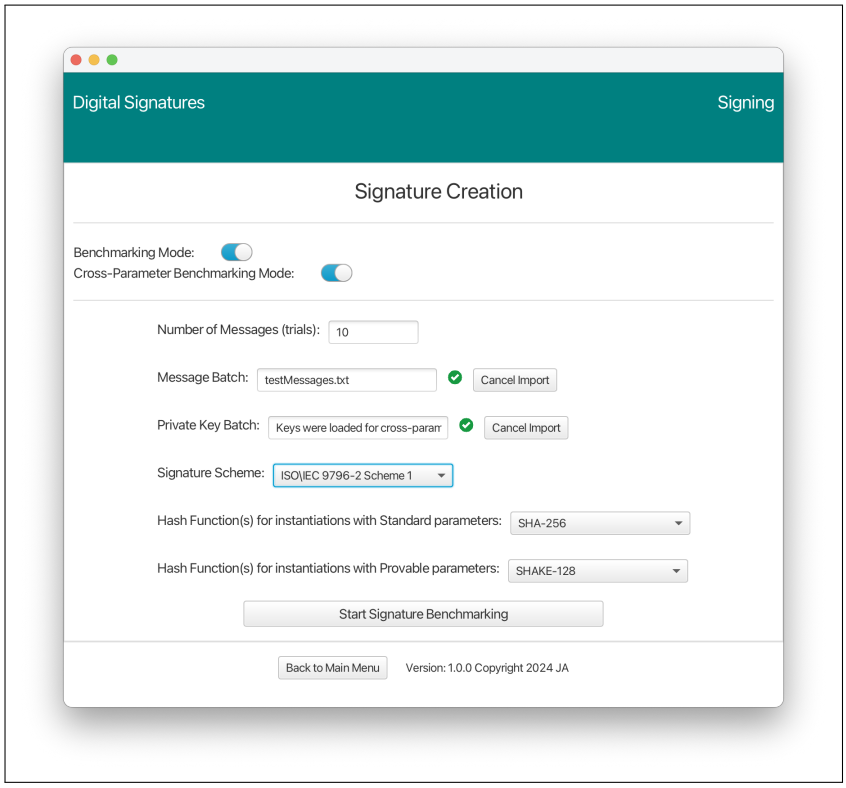


Figure 6.33: Signature Generation with ISO scheme selected

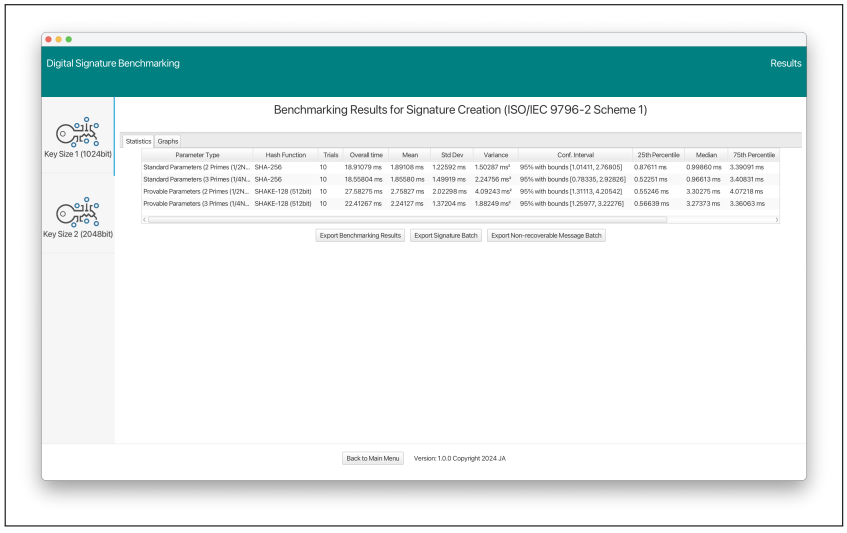


Figure 6.34: Signature Generation Results Screen (with Message Recovery Options)

For signature generation this process involves exporting a batch of non-recoverable messages where each line corresponds with a signature. Lines are prefixed with a "1" to indicate the presence of a non-recoverable message and "0" when there is none. In this specific case, since all messages are too short to produce a non-recoverable part regardless of the key sizes

selected, the resulting file comprises a sequence of "0" flags, each separated by a new line.

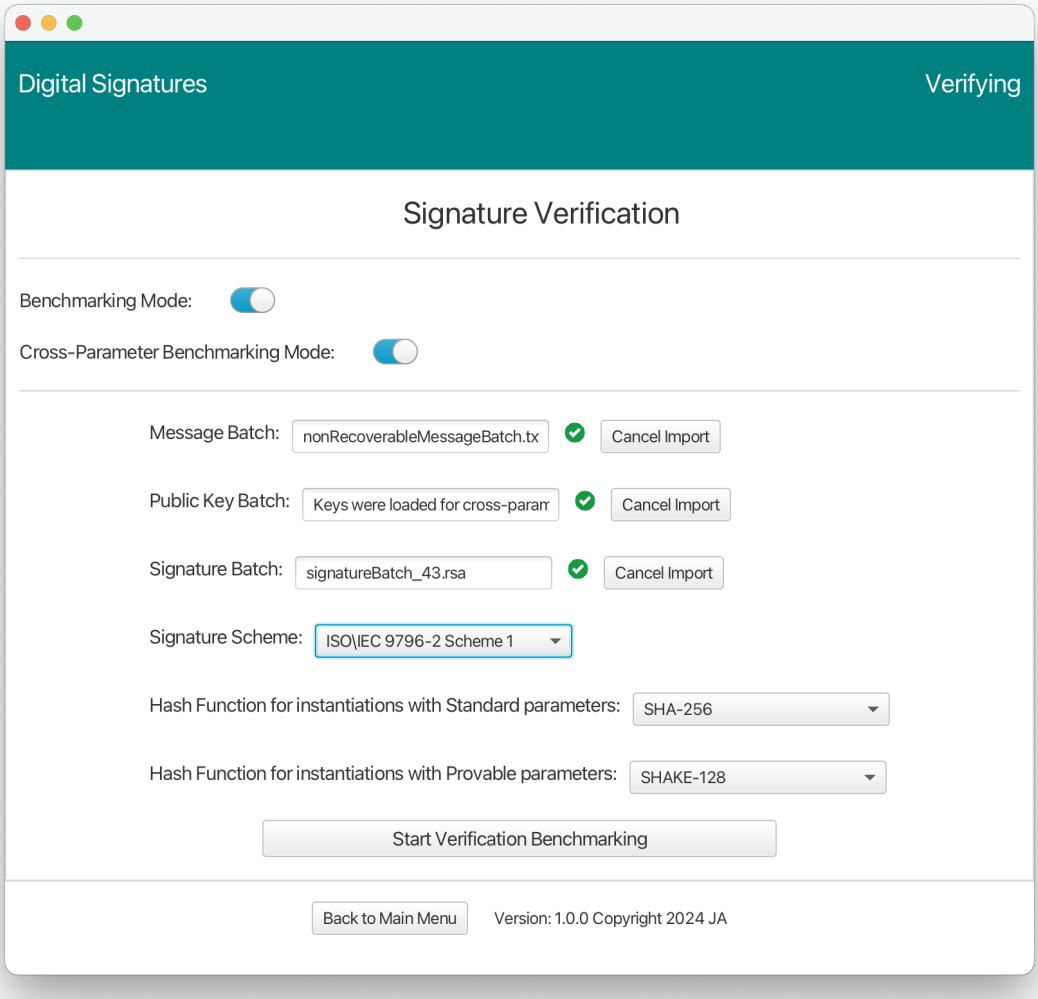


Figure 6.35: Signature verification with ISO scheme selected

The corresponding verification of signatures necessitates import of a non-recoverable message batch file generated from the preceding signature generation benchmarking process. As discussed it contains entries aligned with each signature, organised line by line where each entry starts with a "1" or "0" flag, indicating whether a non-recoverable message part is present or absent, respectively. The application then leverages these non-recoverable message parts, to construct the content of a recovered message batch. This information is then displayed as a dedicated column in the verificationResults CSV files, which users can export for further analysis.

For insight into the testing process see Appendix B.3 for more details.

Chapter 7: **Professional Issues: Adherence to Standards in Cryptography**

A standard is an established set of guidelines or specifications that provide recommendations for processes, practices, or systems. They are internationally agreed ways of working make technology more secure and interoperable [43]. In fact the most common mechanism for determining an appropriate cryptographic primitive, scheme or parameter choice is to consult a standard.

The evolution of cryptography from military to commercial applications created a demand for standardisation in the area. There are thus an extensive range of international standards related to cryptography. Of most pertinence to the signature schemes considered in this project we have the following three:

- ***PKCS# 1 ([6])*** foundational RSA standard that outlines the mathematical properties and structure of RSA public and private keys. It also defines basic algorithms, encoding methods, and padding schemes necessary for RSA primitives.
- ***ANSI X9.31 ([14])*** is a now withdrawn United States financial industry standard for digital signatures based on the RSA algorithm.
- ***ISO/IEC 9796-2 ([15])*** defines three different RSA signature schemes that support message recovery including Digital Signature 1 which is used in the EMV payment system (provides a "platform" for enabling a chip card to be used in banking applications).

The demand for standards in cryptography has supplanted the traditional practice of security through obscurity, where keeping algorithmic details secret was the norm. Today, these details are open to the public, inviting scrutiny and evaluation from the international community of cryptographic researchers. This open approach assures that by the time an algorithm is integrated into a standard, it has withstood the rigours of peer evaluation, instilling confidence in developers who rely on these standards for secure application development.

Standards distill theoretical cryptographic techniques into clear guidelines covering implementation, appropriate contexts for use, and selection of algorithm parameters. They effectively bridge the gap between cryptographers, who typically have a deep understanding of mathematical theory but may lack practical software development experience, and software developers, who often lack extensive knowledge in cryptography.

This delineation of responsibilities ensures that each group can focus on their area of expertise while relying on shared, vetted practices. Standards in cryptography thus serve as a common language, enabling developers to adopt secure algorithms without needing to understand their theoretical underpinnings fully.

The world of cryptography offers numerous instances where implementations found not to follow standards correctly led to critical vulnerabilities [9], [44]–[46]. Among these, the Bleichenbacher signature forgery attacks [10], [28] stands out as a prime example. This class of attacks, which worked on vulnerable implementations of the PKCS#1 signature scheme, showcased the potential for serious security breaches with. A real-world example targeting Facebook [27] in 2018 underscored the risk and during the same period, had been found to be affecting almost a third of the top 100 domains in the Alexa Top 1 Million list. Going further, such was severity of the threat, that a server could potentially be impersonated without accessing its private key. This starkly demonstrate how deviations from standards can lead

to severe security breaches, and emphasises how it should be professional responsibility to implement these standards correctly.

In the context of this project, a key consideration was adhering to the standards defining the implemented signature schemes. It was paramount, not only to ensure that my initial implementations of these schemes conformed to the respective standards delineating them, but also to ensure that any modifications made for incorporating provably secure parameters aligned with established standards.

For example considering the latter, the first change included implementing signatures to use RSA keys with a non-standard number of primes (three instead of two), which the PKCS#1 standard already readily accommodates by defining the modulus N as a product of two or more primes. Another adjustment was the use of a large hash size ($\geq \lambda/2$). Although not directly addressed in the three standards for use in the signature schemes, standardised solutions are available as per the use of variable length hash functions for other use cases. I adopted two of the most established methods for this: one being the MGF1 construction detailed in the PKCS#1 standard [6] and the other being the Keccak-based SHAKE hash functions (standardised by NIST in FIPS 202 [40]), both of which allow for variable length output. These adaptations ensured the project stayed within the bounds of established cryptographic standards.

Furthermore, it was important for me that alignment with standards was to be achieved in a manner that was both ethical and practical. Since cryptographic algorithms are almost always necessary for securing sensitive data that might be communicated or stored, implementation of cryptographic standards, therefore, transcends mere technical expertise. It encompasses a broader professional and ethical responsibility towards safeguarding data and protecting privacy. This responsibility becomes increasingly significant in the current era where data breaches and cyberattacks are commonplace. Thus the approach of attempting to adhere to cryptographic standards I did not view simply a professional mandate but also as an ethical imperative to uphold the integrity and security of digital systems. This holds even in the context of my project which while academic, strived to emulate real-world conditions and implement actual signature schemes. Throughout the project, I was cognisant of these factors and took steps to ensure that my implementations and modifications did not stray significantly from the path charted by the standards.

One example of a hurdle I faced was the foremost one: obtaining the required documentation for the standards related to schemes. Specifically, accessing ANSI X9.31 and ISO/IEC 9796-2 standards was difficult as they are not publicly available, and ISO/IEC standards, in general, are often not freely accessible and can be costly.

Fortunately, the BSI website offers free access to ISO/IEC standards for students at institutions with subscriptions. As Royal Holloway has such a subscription [47], I was able to obtain the ISO/IEC 9796-2 standard. Meanwhile, the PKCS#1 standard was readily available as it is published as part of the Internet Engineering Task Force's (IETF) RFC (Request for comments) series, specifically as RFC 8017 [6].

Addressing the issue with ANSI X9.31, which is a deprecated standard and hence not publicly available, required a different approach. I turned to peer-reviewed academic research papers that detailed the ANSI X9.31 implementation [42]. These papers, while providing more of a high-level overview, aided me building in understanding the scheme. This then allowed me to leverage the similarities between ANSI X9.31 and PKCS#1, both being variants of the signature schemes with appendix class. Using insights provided by the PKCS#1 standard, I was able to address the implementation details that the academic papers did not fully cover. Furthermore the PKCS#1 standard provided a reference for updating the implementation of the ANSI X9.31 scheme to reflect current cryptographic practices not available at the time

the ANSI X9.31 standard was devised.

Chapter 8: **Results and Discussion**

8.1 Setup and Methodology

In order to compare the timing differential between standard and provably secure parameters, multiple disparate benchmarking runs were initiated in comparison benchmarking mode through the key generation portal for 6 key sizes increasing in 1024-bit segments from 1024-bit to 6144-bit. The number of trials for benchmarking was set to 3847 (which is the control variable used for every key configuration across key generation and signature creation/verification) to observe the behaviour of time taken. This specific number is related to the number of trials (messages) in the message batch that was used in benchmarking of the signature schemes.

8.1.1 Dataset

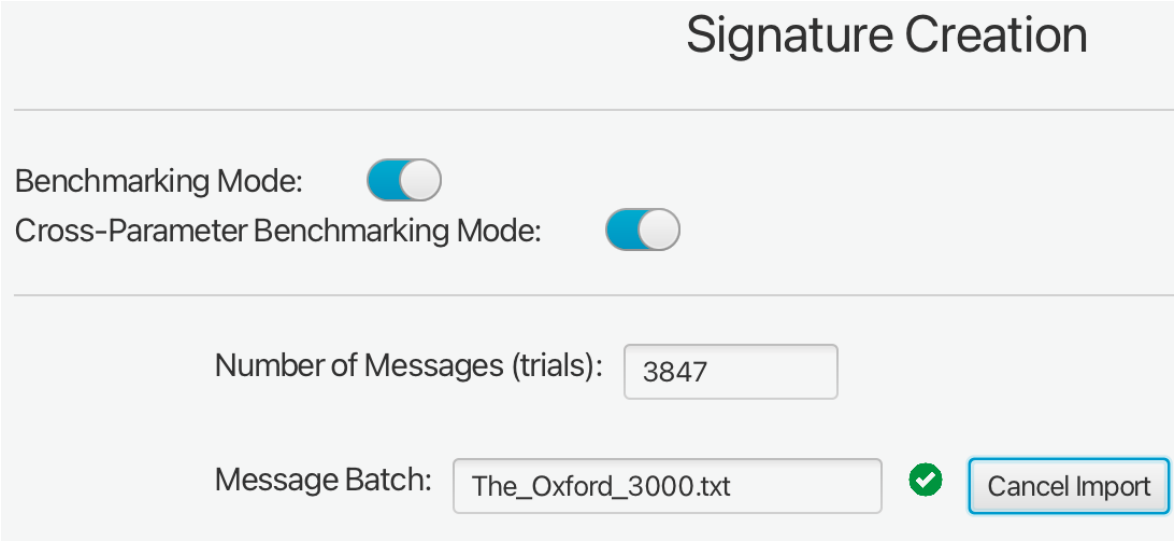


Figure 8.1: Example of Oxford 3000 Dataset being imported for benchmarking

The dataset employed for the benchmarking of Signature Creation and Verification was derived from the first edition of the Oxford 3000 (2017) [48], a collection of the 3847 most important words to learn in English, as defined by the Oxford University Press. This dataset was utilised as the message batch taken from a text file representation of the list [49] publicly available on GitHub. The aim was for each word/line from the Oxford 3000 list to serve as an individual message for processing through the signature schemes. The size of this dataset, with its 3847 entries, served as the control variable applied for every key configuration as its number of trials.

8.1.2 Hash function selection

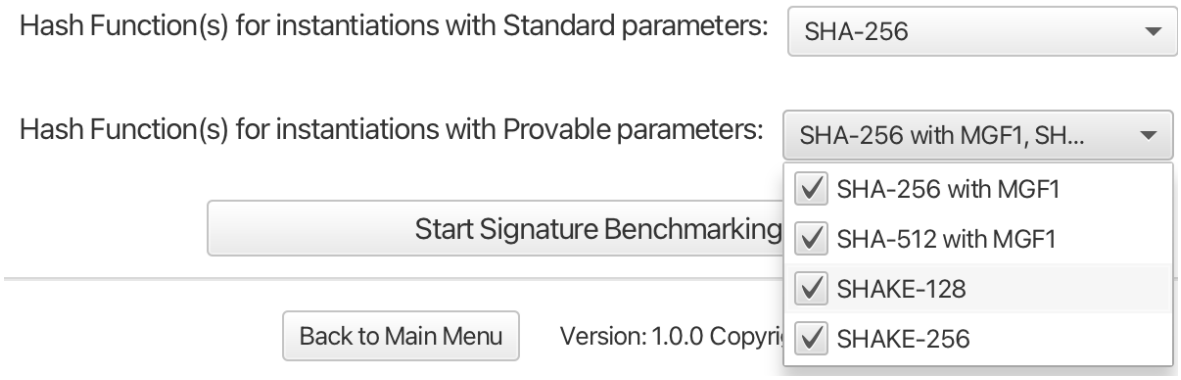


Figure 8.2: Hash function selections respective to each parameter type used for benchmarking

8.1.3 Hardware Specifications

The following hardware specifications outline the computing environment used for all benchmarking exercises:

- **Operating System:** macOS Sonoma 14.3.1
- **CPU:** Apple M1 Pro
- **Total Number of Cores:** 10 (8 performance and 2 efficiency)
- **Clock Speed:** 3220 MHz (and 2064 MHz for Efficiency Cores)
- **Instruction set:** ARMv8-A64 (64 bit)
- **Memory:** 32 GB (LPDDR5 RAM)

It's important to note, that the considered deterministic schemes, often operate in environments vastly different from this testing setup which constitutes a modern computing environment. These include constrained legacy devices or low-performance environments, such as those found in payment/banking systems (e.g., EMV chip-and-pin cards) and secure network protocols. Consequently benchmarking results may not be fully applicable or generalisable to the diverse range of use cases where these older schemes are still operational.

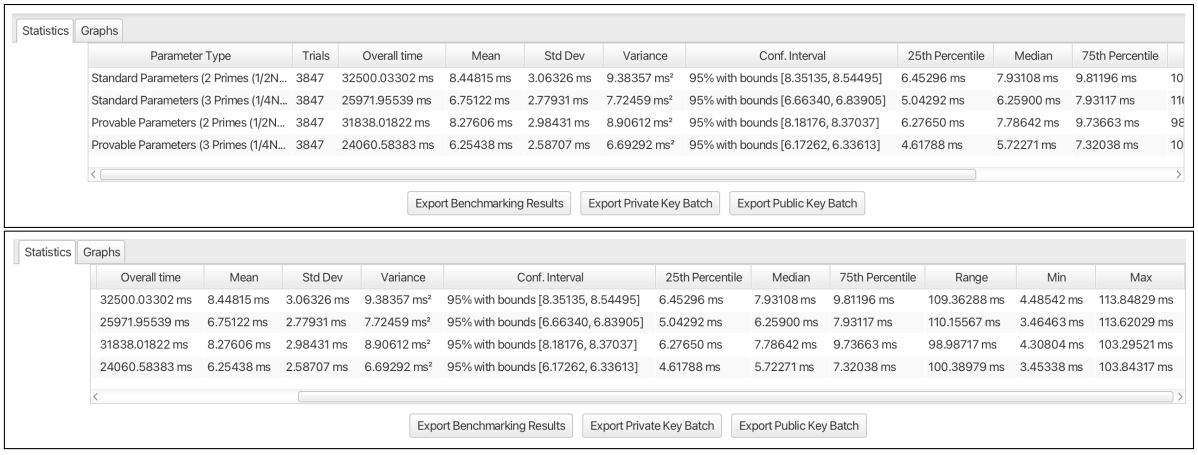
8.2 Results

The results of the timing, using the developed application for key generation and subsequently and signature creation/verification for each deterministic signature scheme is summarised in tables.

**For the views of the complete result tables and subsequent breakdown by parameter type, specific to the ANSI X9.31 rDSA and ISO/IEC 9796-2:2010 Signature Schemes, please consult Appendix D.*

8.3 Key Generation Results

Figure 8.3: Standard vs Provably Secure Key Configurations for 1024-bit Key Size



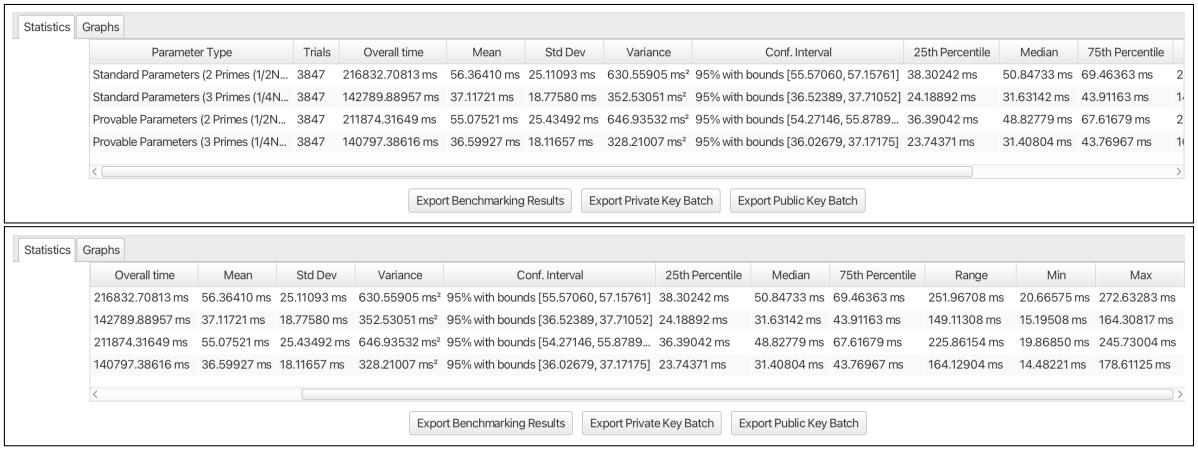
Parameter Type	Trials	Overall time	Mean	Std Dev	Variance	Conf. Interval	25th Percentile	Median	75th Percentile
Standard Parameters (2 Primes (1/2N...	3847	32500.03302 ms	8.44815 ms	3.06326 ms	9.38357 ms²	95% with bounds [8.35135, 8.54495]	6.45296 ms	7.93108 ms	9.81196 ms
Standard Parameters (3 Primes (1/4N...	3847	25971.95539 ms	6.75122 ms	2.77931 ms	7.72459 ms²	95% with bounds [6.66340, 6.83905]	5.04292 ms	6.25900 ms	7.93117 ms
Provable Parameters (2 Primes (1/2N...	3847	31838.01822 ms	8.27606 ms	2.98431 ms	8.90612 ms²	95% with bounds [8.18176, 8.37037]	6.27650 ms	7.78642 ms	9.73663 ms
Provable Parameters (3 Primes (1/4N...	3847	24060.58383 ms	6.25438 ms	2.58707 ms	6.69292 ms²	95% with bounds [6.17262, 6.33613]	4.61788 ms	5.72271 ms	7.32038 ms

Export Benchmarking Results | Export Private Key Batch | Export Public Key Batch

Overall time	Mean	Std Dev	Variance	Conf. Interval	25th Percentile	Median	75th Percentile	Range	Min	Max
32500.03302 ms	8.44815 ms	3.06326 ms	9.38357 ms²	95% with bounds [8.35135, 8.54495]	6.45296 ms	7.93108 ms	9.81196 ms	109.36288 ms	4.48542 ms	113.84829 ms
25971.95539 ms	6.75122 ms	2.77931 ms	7.72459 ms²	95% with bounds [6.66340, 6.83905]	5.04292 ms	6.25900 ms	7.93117 ms	110.15567 ms	3.46463 ms	113.62029 ms
31838.01822 ms	8.27606 ms	2.98431 ms	8.90612 ms²	95% with bounds [8.18176, 8.37037]	6.27650 ms	7.78642 ms	9.73663 ms	98.98717 ms	4.30804 ms	103.29521 ms
24060.58383 ms	6.25438 ms	2.58707 ms	6.69292 ms²	95% with bounds [6.17262, 6.33613]	4.61788 ms	5.72271 ms	7.32038 ms	100.38979 ms	3.45338 ms	103.84317 ms

Export Benchmarking Results | Export Private Key Batch | Export Public Key Batch

Figure 8.4: Standard vs Provably Secure Key Configurations for 2048-bit Key Size



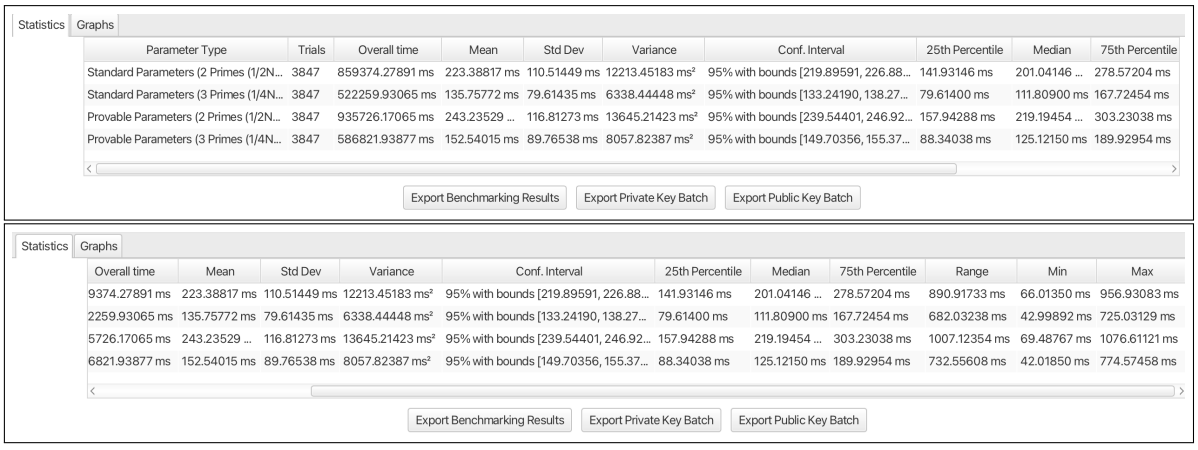
Parameter Type	Trials	Overall time	Mean	Std Dev	Variance	Conf. Interval	25th Percentile	Median	75th Percentile
Standard Parameters (2 Primes (1/2N...	3847	216832.70813 ms	56.36410 ms	25.11093 ms	630.55905 ms²	95% with bounds [55.57060, 57.15761]	38.30242 ms	50.84733 ms	69.46363 ms
Standard Parameters (3 Primes (1/4N...	3847	142789.88957 ms	37.11721 ms	18.77580 ms	352.53051 ms²	95% with bounds [36.52389, 37.71052]	24.18892 ms	31.63142 ms	43.91163 ms
Provable Parameters (2 Primes (1/2N...	3847	211874.31649 ms	55.07521 ms	25.43492 ms	646.93532 ms²	95% with bounds [54.27146, 55.8789...	36.39042 ms	48.82779 ms	67.61679 ms
Provable Parameters (3 Primes (1/4N...	3847	140797.38616 ms	36.59927 ms	18.11657 ms	328.21007 ms²	95% with bounds [36.02679, 37.17175]	23.74371 ms	31.40804 ms	43.76967 ms

Export Benchmarking Results | Export Private Key Batch | Export Public Key Batch

Overall time	Mean	Std Dev	Variance	Conf. Interval	25th Percentile	Median	75th Percentile	Range	Min	Max
216832.70813 ms	56.36410 ms	25.11093 ms	630.55905 ms²	95% with bounds [55.57060, 57.15761]	38.30242 ms	50.84733 ms	69.46363 ms	251.96708 ms	20.66575 ms	272.63283 ms
142789.88957 ms	37.11721 ms	18.77580 ms	352.53051 ms²	95% with bounds [36.52389, 37.71052]	24.18892 ms	31.63142 ms	43.91163 ms	149.11308 ms	15.19508 ms	164.30817 ms
211874.31649 ms	55.07521 ms	25.43492 ms	646.93532 ms²	95% with bounds [54.27146, 55.8789...	36.39042 ms	48.82779 ms	67.61679 ms	225.86154 ms	19.86850 ms	245.73004 ms
140797.38616 ms	36.59927 ms	18.11657 ms	328.21007 ms²	95% with bounds [36.02679, 37.17175]	23.74371 ms	31.40804 ms	43.76967 ms	164.12904 ms	14.48221 ms	178.61125 ms

Export Benchmarking Results | Export Private Key Batch | Export Public Key Batch

Figure 8.5: Standard vs Provably Secure Key Configurations for 3072-bit Key Size



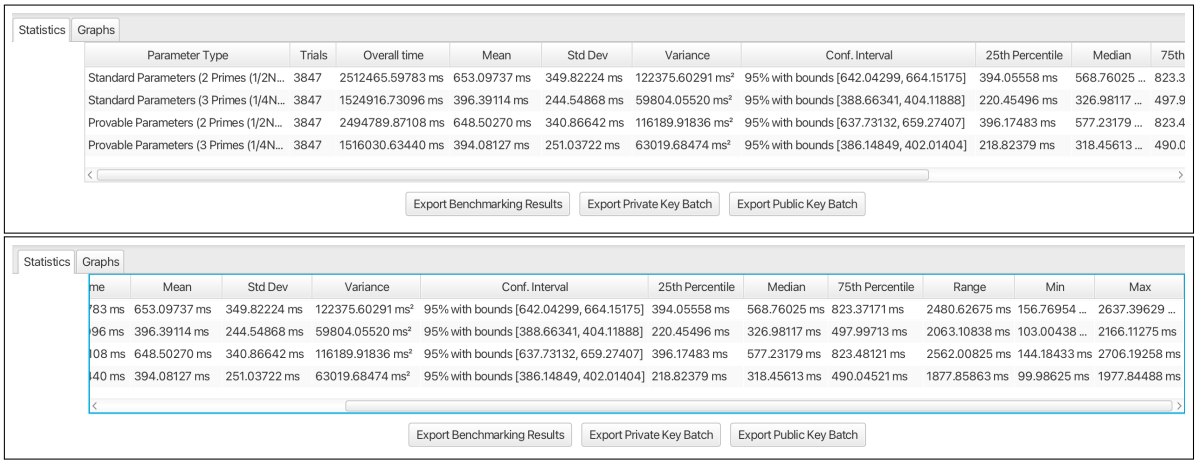
Parameter Type	Trials	Overall time	Mean	Std Dev	Variance	Conf. Interval	25th Percentile	Median	75th Percentile
Standard Parameters (2 Primes (1/2N...	3847	859374.27891 ms	223.38817 ms	110.51449 ms	12213.45183 ms²	95% with bounds [219.89591, 226.88...	141.93146 ms	201.04146 ...	278.57204 ms
Standard Parameters (3 Primes (1/4N...	3847	522259.93065 ms	135.75772 ms	79.61435 ms	6338.44448 ms²	95% with bounds [133.24190, 138.27...	79.61400 ms	111.80900 ms	167.72454 ms
Provable Parameters (2 Primes (1/2N...	3847	935726.17065 ms	243.23529 ...	116.81273 ms	13645.21423 ms²	95% with bounds [239.54401, 246.92...	157.94288 ms	219.19454 ...	303.23038 ms
Provable Parameters (3 Primes (1/4N...	3847	586821.93877 ms	152.54015 ms	89.76538 ms	8057.82387 ms²	95% with bounds [149.70356, 155.37...	88.34038 ms	125.12150 ms	189.92954 ms

Export Benchmarking Results | Export Private Key Batch | Export Public Key Batch

Overall time	Mean	Std Dev	Variance	Conf. Interval	25th Percentile	Median	75th Percentile	Range	Min	Max
9374.27891 ms	223.38817 ms	110.51449 ms	12213.45183 ms²	95% with bounds [219.89591, 226.88...	141.93146 ms	201.04146 ...	278.57204 ms	890.91733 ms	66.01350 ms	956.93083 ms
2259.93065 ms	135.75772 ms	79.61435 ms	6338.44448 ms²	95% with bounds [133.24190, 138.27...	79.61400 ms	111.80900 ms	167.72454 ms	682.03238 ms	42.99892 ms	725.03129 ms
5726.17065 ms	243.23529 ...	116.81273 ms	13645.21423 ms²	95% with bounds [239.54401, 246.92...	157.94288 ms	219.19454 ...	303.23038 ms	1007.12354 ms	69.48767 ms	1076.61121 ms
6821.93877 ms	152.54015 ms	89.76538 ms	8057.82387 ms²	95% with bounds [149.70356, 155.37...	88.34038 ms	125.12150 ms	189.92954 ms	732.55608 ms	42.01850 ms	774.57458 ms

Export Benchmarking Results | Export Private Key Batch | Export Public Key Batch

Figure 8.6: Standard vs Provably Secure Key Configurations for 4096-bit Key Size



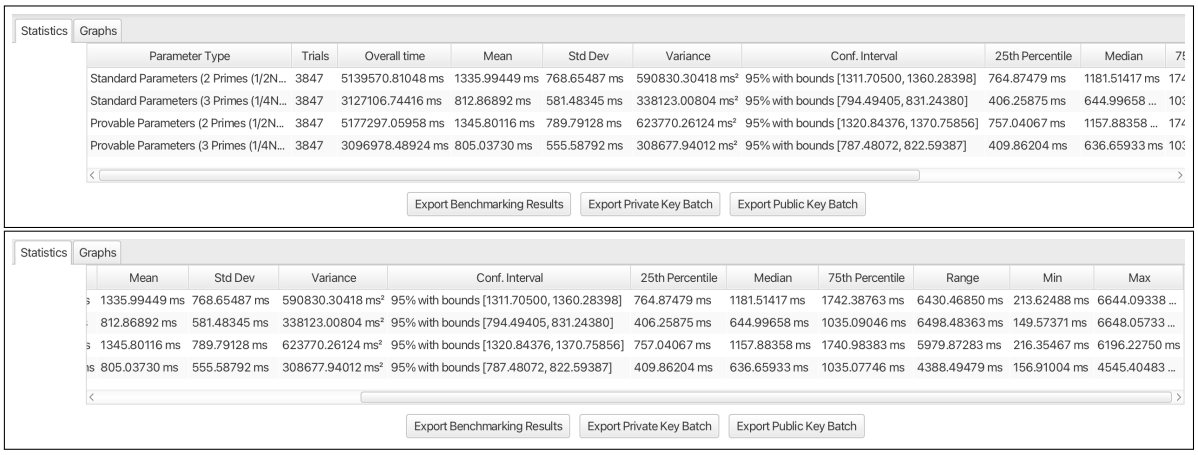
Parameter Type	Trials	Overall time	Mean	Std Dev	Variance	Conf. Interval	25th Percentile	Median	75th
Standard Parameters (2 Primes (1/2N...	3847	2512465.59783 ms	653.09737 ms	349.82224 ms	122375.60291 ms²	95% with bounds [642.04299, 664.15175]	394.05558 ms	568.76025 ...	823.3
Standard Parameters (3 Primes (1/4N...	3847	1524916.73096 ms	396.39114 ms	244.54868 ms	59804.05520 ms²	95% with bounds [388.66341, 404.11888]	220.45496 ms	326.98117 ...	497.9
Provable Parameters (2 Primes (1/2N...	3847	2494789.87108 ms	648.50270 ms	340.86642 ms	116189.91836 ms²	95% with bounds [637.73132, 659.27407]	396.17483 ms	577.23179 ...	823.4
Provable Parameters (3 Primes (1/4N...	3847	1516030.63440 ms	394.08127 ms	251.03722 ms	63019.68474 ms²	95% with bounds [386.14849, 402.01404]	218.82379 ms	318.45613 ...	490.0

Export Benchmarking Results | Export Private Key Batch | Export Public Key Batch

me	Mean	Std Dev	Variance	Conf. Interval	25th Percentile	Median	75th Percentile	Range	Min	Max
783 ms	653.09737 ms	349.82224 ms	122375.60291 ms²	95% with bounds [642.04299, 664.15175]	394.05558 ms	568.76025 ms	823.37171 ms	2480.62675 ms	156.76954 ...	2637.39629 ...
396 ms	396.39114 ms	244.54868 ms	59804.05520 ms²	95% with bounds [388.66341, 404.11888]	220.45496 ms	326.98117 ms	497.99713 ms	2063.10838 ms	103.00438 ...	2166.11275 ms
108 ms	648.50270 ms	340.86642 ms	116189.91836 ms²	95% with bounds [637.73132, 659.27407]	396.17483 ms	577.23179 ms	823.48121 ms	2562.00825 ms	144.18433 ms	2706.19258 ms
140 ms	394.08127 ms	251.03722 ms	63019.68474 ms²	95% with bounds [386.14849, 402.01404]	218.82379 ms	318.45613 ms	490.04521 ms	1877.85863 ms	99.98625 ms	1977.84488 ms

Export Benchmarking Results | Export Private Key Batch | Export Public Key Batch

Figure 8.7: Standard vs Provably Secure Key Configurations for 5120-bit Key Size



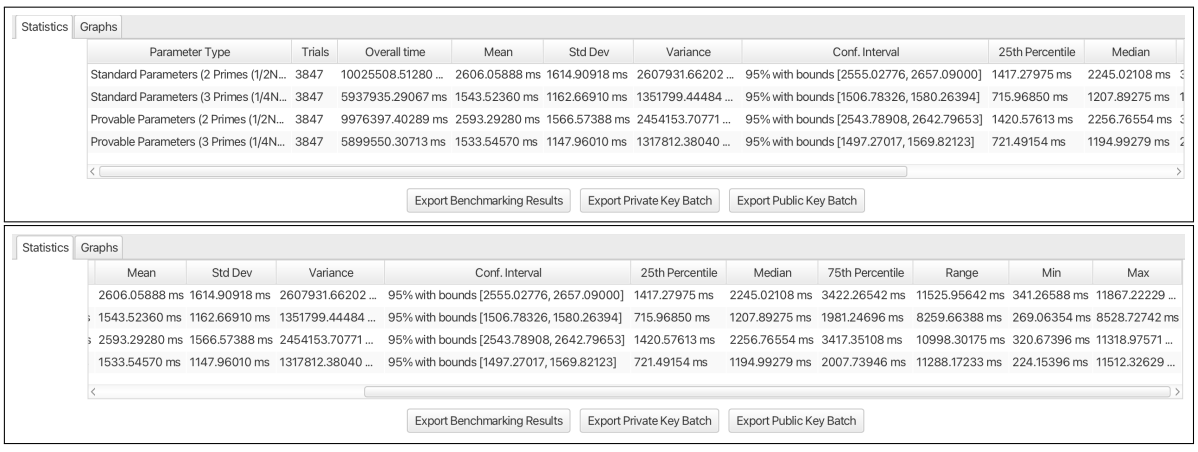
Parameter Type	Trials	Overall time	Mean	Std Dev	Variance	Conf. Interval	25th Percentile	Median	75
Standard Parameters (2 Primes (1/2N...	3847	5139570.81048 ms	1335.99449 ms	768.65487 ms	590830.30418 ms²	95% with bounds [1311.70500, 1360.28398]	764.87479 ms	1181.51417 ms	174
Standard Parameters (3 Primes (1/4N...	3847	3127106.74416 ms	812.86892 ms	581.48345 ms	338123.00804 ms²	95% with bounds [794.49405, 831.24380]	406.25875 ms	644.99658 ...	103
Provable Parameters (2 Primes (1/2N...	3847	5177297.05958 ms	1345.80116 ms	789.79128 ms	623770.26124 ms²	95% with bounds [1320.84376, 1370.75856]	757.04067 ms	1157.88358 ...	174
Provable Parameters (3 Primes (1/4N...	3847	3096978.48924 ms	805.03730 ms	555.58792 ms	308677.94012 ms²	95% with bounds [787.48072, 822.59387]	409.86204 ms	636.65933 ms	103

Export Benchmarking Results | Export Private Key Batch | Export Public Key Batch

Mean	Std Dev	Variance	Conf. Interval	25th Percentile	Median	75th Percentile	Range	Min	Max
1335.99449 ms	768.65487 ms	590830.30418 ms²	95% with bounds [1311.70500, 1360.28398]	764.87479 ms	1181.51417 ms	1742.38763 ms	6430.46850 ms	213.62488 ms	6644.09338 ...
812.86892 ms	581.48345 ms	338123.00804 ms²	95% with bounds [794.49405, 831.24380]	406.25875 ms	644.99658 ms	1035.09046 ms	6498.48363 ms	149.57371 ms	6648.05733 ...
1345.80116 ms	789.79128 ms	623770.26124 ms²	95% with bounds [1320.84376, 1370.75856]	757.04067 ms	1157.88358 ms	1740.98383 ms	5979.87283 ms	216.35467 ms	6196.22750 ms
805.03730 ms	555.58792 ms	308677.94012 ms²	95% with bounds [787.48072, 822.59387]	409.86204 ms	636.65933 ms	1035.07746 ms	4388.49479 ms	156.91004 ms	4545.40483 ...

Export Benchmarking Results | Export Private Key Batch | Export Public Key Batch

Figure 8.8: Standard vs Provably Secure Key Configurations for 6144-bit Key Size



Parameter Type	Trials	Overall time	Mean	Std Dev	Variance	Conf. Interval	25th Percentile	Median
Standard Parameters (2 Primes (1/2N...	3847	10025508.51280 ...	2606.05888 ms	1614.90918 ms	2607931.66202 ...	95% with bounds [2555.02776, 2657.09000]	1417.27975 ms	2245.02108 ms
Standard Parameters (3 Primes (1/4N...	3847	5937935.29067 ms	1543.52360 ms	1162.66910 ms	1351799.44484 ...	95% with bounds [1506.78326, 1580.26394]	715.96850 ms	1207.89275 ms
Provable Parameters (2 Primes (1/2N...	3847	9976397.40289 ms	2593.29280 ms	1566.57388 ms	2454153.70771 ...	95% with bounds [2543.78908, 2642.79653]	1420.57613 ms	2256.76554 ms
Provable Parameters (3 Primes (1/4N...	3847	5899550.30713 ms	1533.54570 ms	1147.96010 ms	1317812.38040 ...	95% with bounds [1497.27017, 1569.82123]	721.49154 ms	1194.99279 ms

Export Benchmarking Results | Export Private Key Batch | Export Public Key Batch

Mean	Std Dev	Variance	Conf. Interval	25th Percentile	Median	75th Percentile	Range	Min	Max
2606.05888 ms	1614.90918 ms	2607931.66202 ...	95% with bounds [2555.02776, 2657.09000]	1417.27975 ms	2245.02108 ms	3422.26542 ms	11525.95642 ms	341.26588 ms	11867.22229 ...
1543.52360 ms	1162.66910 ms	1351799.44484 ...	95% with bounds [1506.78326, 1580.26394]	715.96850 ms	1207.89275 ms	1981.24696 ms	8259.66388 ms	269.06354 ms	8528.72742 ms
2593.29280 ms	1566.57388 ms	2454153.70771 ...	95% with bounds [2543.78908, 2642.79653]	1420.57613 ms	2256.76554 ms	3417.35108 ms	10998.30175 ms	320.67396 ms	11318.97571 ...
1533.54570 ms	1147.96010 ms	1317812.38040 ...	95% with bounds [1497.27017, 1569.82123]	721.49154 ms	1194.99279 ms	2007.73946 ms	11288.17233 ms	224.15396 ms	11512.32629 ...

Export Benchmarking Results | Export Private Key Batch | Export Public Key Batch

Tables 8.3 to 8.8 show timing statistics in milliseconds (correct to 5 decimal places) for both standard and provably secure key configurations across a range of key sizes, incrementing by 1024 bits from 1024 to 6144 bits.

Comparing 2 vs 3 primes within parameter groups

Table 8.1 illustrates the percentage difference in mean generation times when moving from 2-prime to 3-prime configurations for each parameter group.

Table 8.1: Efficiency Comparison of 2 vs 3 Primes in Key Generation (Percentage Difference)

Key Size (bits)	**2 Primes to 3 Primes % Diff (Standard Parameters)**	**2 Primes to 3 Primes % Diff (Provably Secure Parameters)**
1024	-20.07%	-24.43%
2048	-34.17%	-33.54%
3072	-39.22%	-37.30%
4096	-39.30%	-39.23%
5120	-39.16%	-40.20%
6144	-40.77%	-40.85%

A consistent trend observed across all key sizes from 1024 to 6144 bits is the superior performance of 3-prime configurations over 2-prime configurations. This pattern holds true within both standard and provably secure parameter types. Notably, the efficiency advantage of 3-prime configurations also increased with key size.

Comparing Standard vs provably secure key configurations

The distinction of the pattern being within parameter types is an important one to make because it illustrates what appears to be an inherent efficiency gain in key generation when additional prime factors are used, irrespective of whether the parameters are standard or provably secure. For example, at the 1024-bit key size, while both provably secure configurations with 2 and 3 primes show a slight performance increase over their standard counterparts, comparing across different parameter types with a different number of primes (like 2 primes provably secure vs. 3 primes standard) is not equitable due to the fundamental difference in the configuration setup.

More concretely, at the 1024-bit key size 8.3, provably secure configurations with both 2 and 3 primes show modest efficiency gains over standard counterparts. Specifically, the provably secure 2-prime configuration was approximately 2.04% quicker (8.27606 ms compared to 8.44815 ms), and the provably secure 3-prime configurations demonstrated a 7.35% reduction in mean time (6.25438 ms versus 6.75122 ms). This trend challenges the usual belief that enhanced security compromises performance. Both configuration types exhibit relatively lower standard deviation and variance at this size, indicating consistent performance.

As key sizes increase, to 2048 bits 8.4, the pattern of improved efficiency in provably secure configurations compared to standard holds, with about 2% faster mean time in the 2-prime configuration and 1.4% in 3-prime configuration. However, at 3072 bits 8.5, a deviation emerges: while the 3-prime provably secure configuration maintained a 12.3% efficiency advantage, the 2-prime provably secure configuration bucked the trend coming in at 8.91% slower than its standard counterpart. This size also marked a notable increase in standard deviation and variance, a suggestion of a wider spread in key generation times.

Moving to 4096-bit and 5120-bit key sizes, provably secure configurations maintain their lead in efficiency over standard configurations, although the margin gets narrower.

At the largest tested size of 6144 bits 8.8, the efficiency gap remains minimal, with provably secure parameters showing only about 0.49% and 0.64% improvements for 2-prime and 3-prime configurations respectively. This key size experienced the highest standard deviation and variance, emphasising significant variability in key generation times.

Summary

Throughout the analysis, as expected, the time taken for key generation increases with the key size. However, the increase is not linear; it tends to grow more than proportionally, reflecting the greater computational complexity involved in generating and handling larger primes. Particularly notable in this context is the significant escalation in standard deviation and variance at larger key sizes for both standard and provably secure configurations.

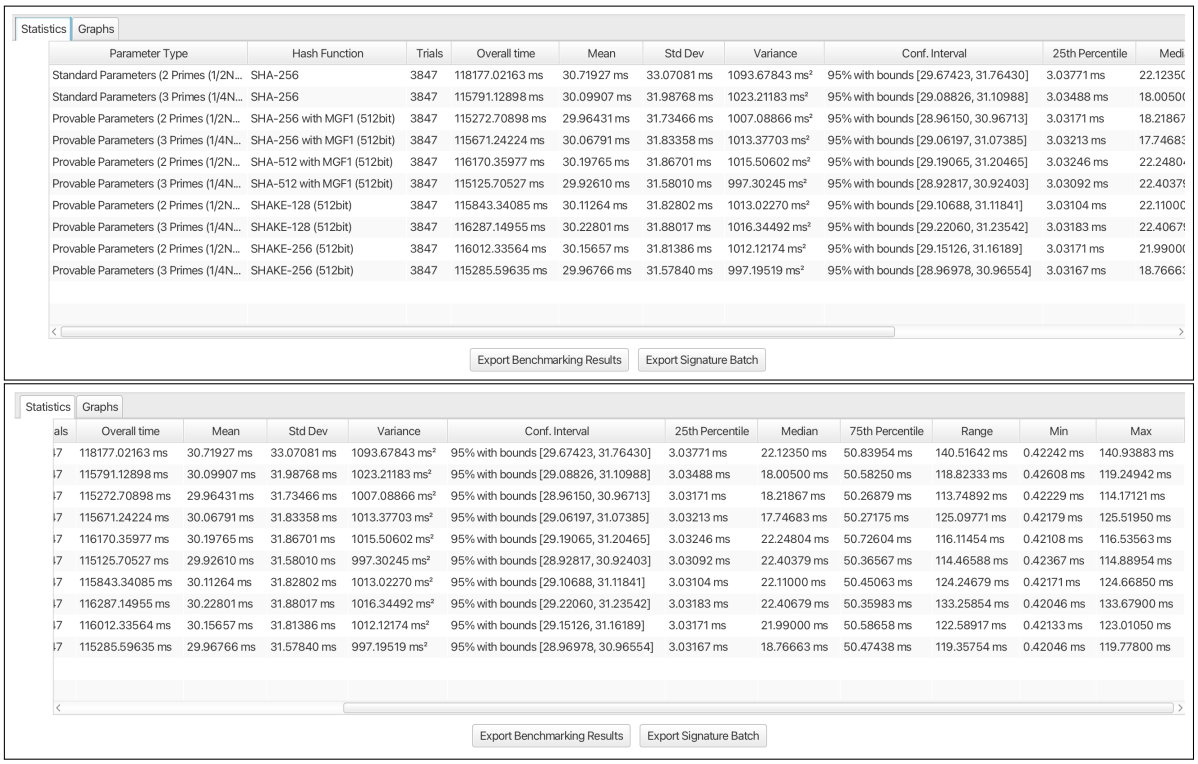
While provably secure parameters exhibit some efficiency improvements, these gains are relatively minimal, especially as key sizes grow larger. The increased variability at larger key sizes, as indicated by higher standard deviation and variance, obscures the true impact of opting for provably secure parameters since the small percentage difference may fall within the broader range of natural variability of the key generation process. Consequently, although provably secure configurations show efficiency improvements at smaller key sizes, their practical significance wanes with increasing key sizes.

In essence there is no significant difference between key generation using standard versus provably secure configurations and the distinction between the two becomes negligible, with increasingly larger key sizes. This observation challenges the expectation that using provably secure parameters would impede performance. It implies that adopting provably secure key configurations might not only be a security upgrade but also a practically viable choice that in some cases offers an improvement in performance.

8.4 Signature Creation Results

8.4.1 Signature Creation Results (PKCS#1 v1.5)

Figure 8.9: Instantiation of PKCS#1 v1.5 with standard vs provably secure parameters (1024-bit Key Size) for signature creation



Statistics | Graphs

Parameter Type	Hash Function	Trials	Overall time	Mean	Std Dev	Variance	Conf. Interval	25th Percentile	Medi
Standard Parameters (2 Primes (1/2N...	SHA-256	3847	118177.02163 ms	30.71927 ms	33.07081 ms	1093.67843 ms²	95% with bounds [29.67423, 31.76430]	3.03771 ms	22.1235(
Standard Parameters (3 Primes (1/4N...	SHA-256	3847	115791.12898 ms	30.09907 ms	31.98768 ms	1023.21183 ms²	95% with bounds [29.08826, 31.10988]	3.03488 ms	18.0050(
Provable Parameters (2 Primes (1/2N...	SHA-256 with MGF1 (512bit)	3847	115272.70898 ms	29.96431 ms	31.73466 ms	1007.08866 ms²	95% with bounds [28.96150, 30.96713]	3.03171 ms	18.21867
Provable Parameters (3 Primes (1/4N...	SHA-256 with MGF1 (512bit)	3847	115671.24224 ms	30.06791 ms	31.83358 ms	1013.37703 ms²	95% with bounds [29.06197, 31.07385]	3.03213 ms	17.74683
Provable Parameters (2 Primes (1/2N...	SHA-512 with MGF1 (512bit)	3847	116170.35977 ms	30.19765 ms	31.86701 ms	1015.50602 ms²	95% with bounds [29.19065, 31.20465]	3.03246 ms	22.2480
Provable Parameters (3 Primes (1/4N...	SHA-512 with MGF1 (512bit)	3847	115125.70527 ms	29.92610 ms	31.58010 ms	997.30245 ms²	95% with bounds [28.92817, 30.92403]	3.03092 ms	22.4037
Provable Parameters (2 Primes (1/2N...	SHAKE-128 (512bit)	3847	115843.34085 ms	30.11264 ms	31.82802 ms	1013.02270 ms²	95% with bounds [29.10688, 31.11841]	3.03104 ms	22.11000
Provable Parameters (3 Primes (1/4N...	SHAKE-128 (512bit)	3847	116287.14955 ms	30.22801 ms	31.88017 ms	1016.34492 ms²	95% with bounds [29.22060, 31.23542]	3.03183 ms	22.4067
Provable Parameters (2 Primes (1/2N...	SHAKE-256 (512bit)	3847	116012.33564 ms	30.15657 ms	31.81386 ms	1012.12174 ms²	95% with bounds [29.15126, 31.16189]	3.03171 ms	21.9900(
Provable Parameters (3 Primes (1/4N...	SHAKE-256 (512bit)	3847	115285.59635 ms	29.96766 ms	31.57840 ms	997.19519 ms²	95% with bounds [28.96978, 30.96554]	3.03167 ms	18.7666

Export Benchmarking Results | Export Signature Batch

Statistics | Graphs

als	Overall time	Mean	Std Dev	Variance	Conf. Interval	25th Percentile	Median	75th Percentile	Range	Min	Max
47	118177.02163 ms	30.71927 ms	33.07081 ms	1093.67843 ms²	95% with bounds [29.67423, 31.76430]	3.03771 ms	22.12350 ms	50.83954 ms	140.51642 ms	0.42242 ms	140.93883 ms
47	115791.12898 ms	30.09907 ms	31.98768 ms	1023.21183 ms²	95% with bounds [29.08826, 31.10988]	3.03488 ms	18.00500 ms	50.58250 ms	118.82333 ms	0.42608 ms	119.24942 ms
47	115272.70898 ms	29.96431 ms	31.73466 ms	1007.08866 ms²	95% with bounds [28.96150, 30.96713]	3.03171 ms	18.21867 ms	50.26879 ms	113.74892 ms	0.42229 ms	114.17121 ms
47	115671.24224 ms	30.06791 ms	31.83358 ms	1013.37703 ms²	95% with bounds [29.06197, 31.07385]	3.03213 ms	17.74683 ms	50.27175 ms	125.09771 ms	0.42179 ms	125.51950 ms
47	116170.35977 ms	30.19765 ms	31.86701 ms	1015.50602 ms²	95% with bounds [29.19065, 31.20465]	3.03246 ms	22.24804 ms	50.72604 ms	116.11454 ms	0.42108 ms	116.53563 ms
47	115125.70527 ms	29.92610 ms	31.58010 ms	997.30245 ms²	95% with bounds [28.92817, 30.92403]	3.03092 ms	22.40379 ms	50.36567 ms	114.46588 ms	0.42367 ms	114.88954 ms
47	115843.34085 ms	30.11264 ms	31.82802 ms	1013.02270 ms²	95% with bounds [29.10688, 31.11841]	3.03104 ms	22.11000 ms	50.45063 ms	124.24679 ms	0.42171 ms	124.66850 ms
47	116287.14955 ms	30.22801 ms	31.88017 ms	1016.34492 ms²	95% with bounds [29.22060, 31.23542]	3.03183 ms	22.40679 ms	50.35983 ms	133.25854 ms	0.42046 ms	133.67900 ms
47	116012.33564 ms	30.15657 ms	31.81386 ms	1012.12174 ms²	95% with bounds [29.15126, 31.16189]	3.03171 ms	21.99000 ms	50.58658 ms	122.58917 ms	0.42133 ms	123.01050 ms
47	115285.59635 ms	29.96766 ms	31.57840 ms	997.19519 ms²	95% with bounds [28.96978, 30.96554]	3.03167 ms	18.76663 ms	50.47438 ms	119.35754 ms	0.42046 ms	119.77800 ms

Export Benchmarking Results | Export Signature Batch

Figure 8.10: Instantiation of PKCS#1 v1.5 with standard vs provably secure parameters (2048-bit Key Size) for signature creation

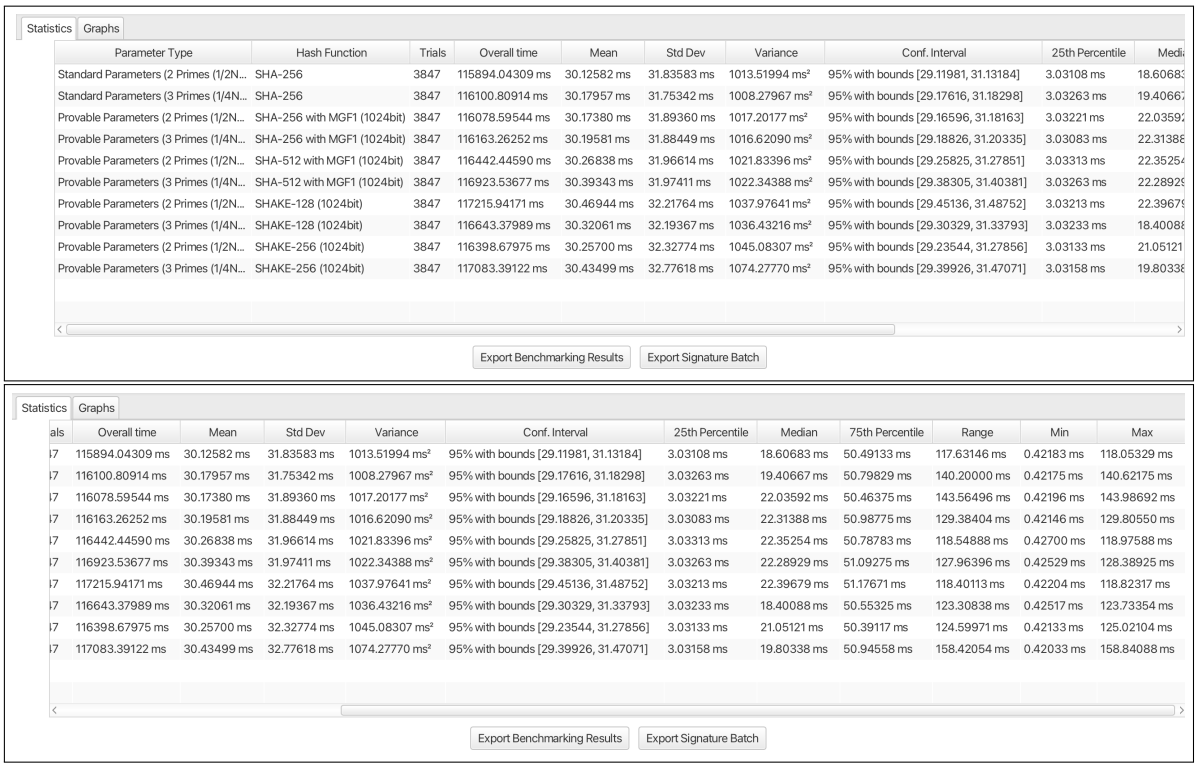


Figure 8.11: Instantiation of PKCS#1 v1.5 with standard vs provably secure parameters (3072-bit Key Size) for signature creation

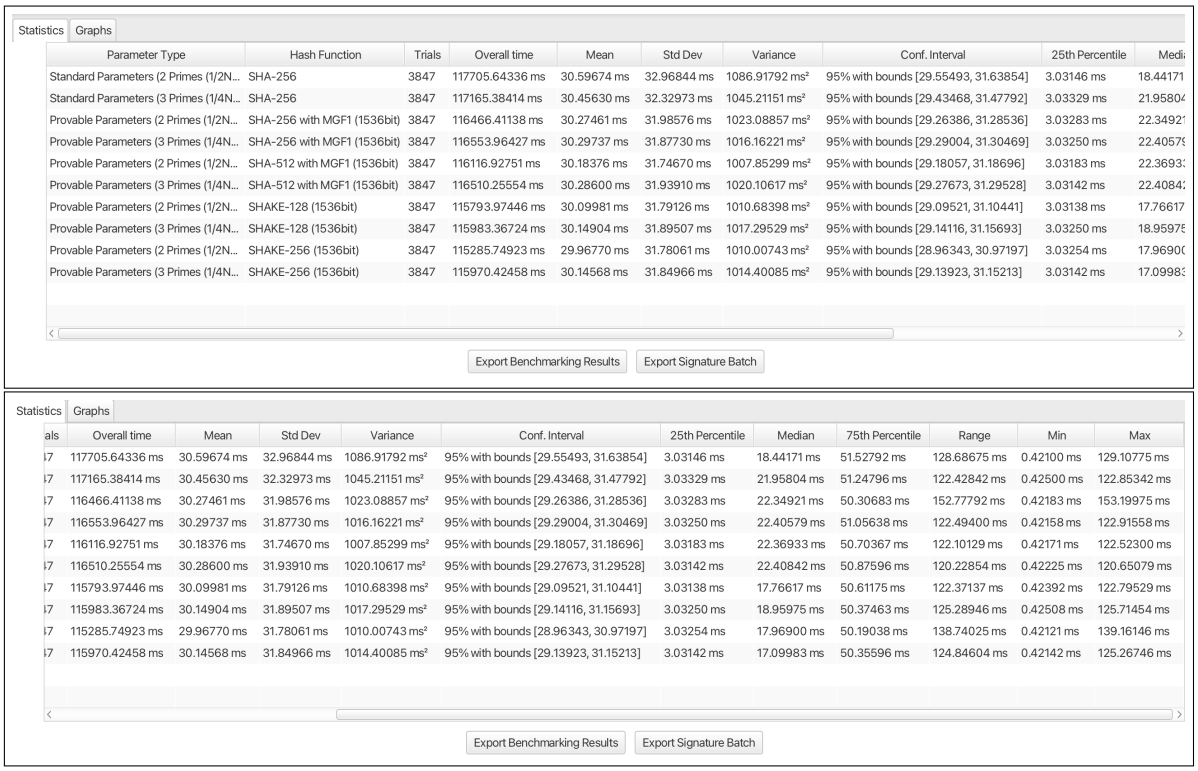
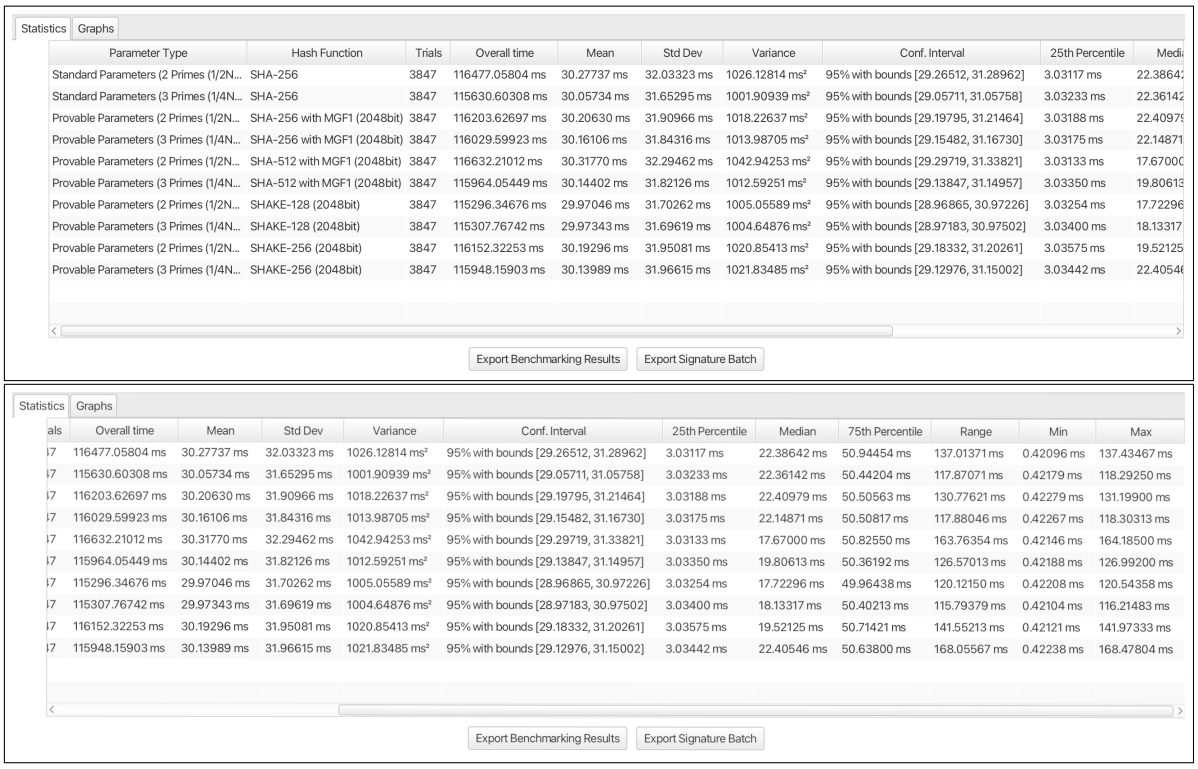


Figure 8.12: Instantiation of PKCS#1 v1.5 with standard vs provably secure parameters (4096-bit Key Size) for signature creation



Statistics | Graphs

Parameter Type	Hash Function	Trials	Overall time	Mean	Std Dev	Variance	Conf. Interval	25th Percentile	Medi
Standard Parameters (2 Primes (1/2N...	SHA-256	3847	116477.05804 ms	30.27737 ms	32.03323 ms	1026.12814 ms²	95% with bounds [29.26512, 31.28962]	3.03117 ms	22.3864
Standard Parameters (3 Primes (1/4N...	SHA-256	3847	115630.60308 ms	30.05734 ms	31.65295 ms	1001.90939 ms²	95% with bounds [29.05711, 31.05758]	3.03233 ms	22.3614
Provable Parameters (2 Primes (1/2N...	SHA-256 with MGF1 (2048bit)	3847	116203.62697 ms	30.20630 ms	31.90966 ms	1018.22637 ms²	95% with bounds [29.19795, 31.21464]	3.03188 ms	22.4097
Provable Parameters (3 Primes (1/4N...	SHA-256 with MGF1 (2048bit)	3847	116029.59923 ms	30.16106 ms	31.84316 ms	1013.98705 ms²	95% with bounds [29.15482, 31.16730]	3.03175 ms	22.1487
Provable Parameters (2 Primes (1/2N...	SHA-512 with MGF1 (2048bit)	3847	116632.21012 ms	30.31770 ms	32.29462 ms	1042.94253 ms²	95% with bounds [29.29719, 31.33821]	3.03133 ms	17.6700
Provable Parameters (3 Primes (1/4N...	SHA-512 with MGF1 (2048bit)	3847	115964.05449 ms	30.14402 ms	31.82126 ms	1012.59251 ms²	95% with bounds [29.13847, 31.14957]	3.03350 ms	19.8061
Provable Parameters (2 Primes (1/2N...	SHAKE-128 (2048bit)	3847	115296.34676 ms	29.97046 ms	31.70262 ms	1005.05589 ms²	95% with bounds [28.96865, 30.97226]	3.03254 ms	17.7229
Provable Parameters (3 Primes (1/4N...	SHAKE-128 (2048bit)	3847	115307.76742 ms	29.97343 ms	31.69619 ms	1004.64876 ms²	95% with bounds [28.97183, 30.97502]	3.03400 ms	18.1331
Provable Parameters (2 Primes (1/2N...	SHAKE-256 (2048bit)	3847	116152.32253 ms	30.19296 ms	31.95081 ms	1020.85413 ms²	95% with bounds [29.18332, 31.20261]	3.03575 ms	19.5212
Provable Parameters (3 Primes (1/4N...	SHAKE-256 (2048bit)	3847	115948.15903 ms	30.13989 ms	31.96615 ms	1021.83485 ms²	95% with bounds [29.12976, 31.15002]	3.03442 ms	22.4054

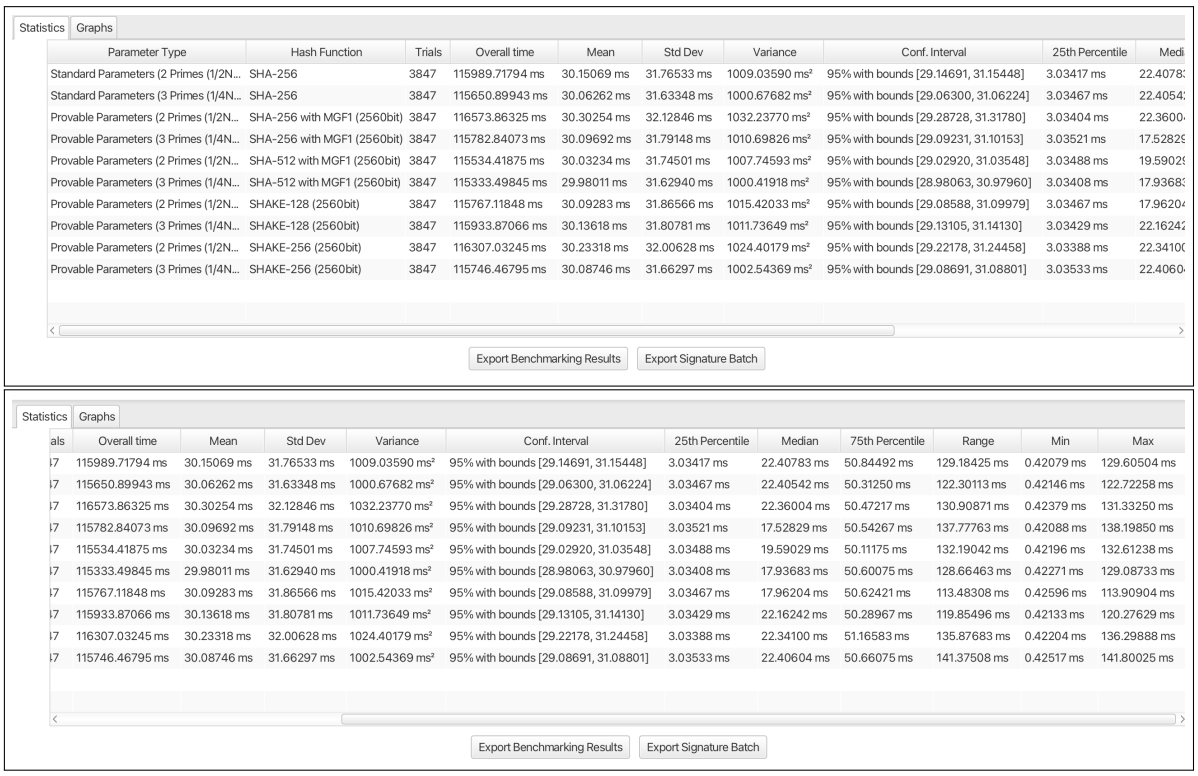
Export Benchmarking Results | Export Signature Batch

Statistics | Graphs

als	Overall time	Mean	Std Dev	Variance	Conf. Interval	25th Percentile	Median	75th Percentile	Range	Min	Max
7	116477.05804 ms	30.27737 ms	32.03323 ms	1026.12814 ms²	95% with bounds [29.26512, 31.28962]	3.03117 ms	22.38642 ms	50.94454 ms	137.01371 ms	0.42096 ms	137.43467 ms
7	115630.60308 ms	30.05734 ms	31.65295 ms	1001.90939 ms²	95% with bounds [29.05711, 31.05758]	3.03233 ms	22.36142 ms	50.44204 ms	117.87071 ms	0.42179 ms	118.29250 ms
7	116203.62697 ms	30.20630 ms	31.90966 ms	1018.22637 ms²	95% with bounds [29.19795, 31.21464]	3.03188 ms	22.40979 ms	50.50563 ms	130.77621 ms	0.42279 ms	131.19900 ms
7	116029.59923 ms	30.16106 ms	31.84316 ms	1013.98705 ms²	95% with bounds [29.15482, 31.16730]	3.03175 ms	22.14871 ms	50.50817 ms	117.88046 ms	0.42267 ms	118.30313 ms
7	116632.21012 ms	30.31770 ms	32.29462 ms	1042.94253 ms²	95% with bounds [29.29719, 31.33821]	3.03133 ms	17.67000 ms	50.82550 ms	163.76354 ms	0.42146 ms	164.18500 ms
7	115964.05449 ms	30.14402 ms	31.82126 ms	1012.59251 ms²	95% with bounds [29.13847, 31.14957]	3.03350 ms	19.80613 ms	50.36192 ms	126.57013 ms	0.42188 ms	126.99200 ms
7	115296.34676 ms	29.97046 ms	31.70262 ms	1005.05589 ms²	95% with bounds [28.96865, 30.97226]	3.03254 ms	17.72296 ms	49.96438 ms	120.12150 ms	0.42208 ms	120.54358 ms
7	115307.76742 ms	29.97343 ms	31.69619 ms	1004.64876 ms²	95% with bounds [28.97183, 30.97502]	3.03400 ms	18.13317 ms	50.40213 ms	115.79379 ms	0.42104 ms	116.21483 ms
7	116152.32253 ms	30.19296 ms	31.95081 ms	1020.85413 ms²	95% with bounds [29.18332, 31.20261]	3.03575 ms	19.52125 ms	50.71421 ms	141.55213 ms	0.42121 ms	141.97333 ms
7	115948.15903 ms	30.13989 ms	31.96615 ms	1021.83485 ms²	95% with bounds [29.12976, 31.15002]	3.03442 ms	22.40546 ms	50.63800 ms	168.05567 ms	0.42238 ms	168.47804 ms

Export Benchmarking Results | Export Signature Batch

Figure 8.13: Instantiation of PKCS#1 v1.5 with standard vs provably secure parameters (5120-bit Key Size) for signature creation



Statistics | Graphs

Parameter Type	Hash Function	Trials	Overall time	Mean	Std Dev	Variance	Conf. Interval	25th Percentile	Medi
Standard Parameters (2 Primes (1/2N...	SHA-256	3847	115989.71794 ms	30.15069 ms	31.76533 ms	1009.03590 ms²	95% with bounds [29.14691, 31.15448]	3.03417 ms	22.4078
Standard Parameters (3 Primes (1/4N...	SHA-256	3847	115650.89943 ms	30.06262 ms	31.63348 ms	1000.67682 ms²	95% with bounds [29.06300, 31.06224]	3.03467 ms	22.4054
Provable Parameters (2 Primes (1/2N...	SHA-256 with MGF1 (2560bit)	3847	116573.86325 ms	30.30254 ms	32.12846 ms	1032.23770 ms²	95% with bounds [29.28728, 31.31780]	3.03404 ms	22.3600
Provable Parameters (3 Primes (1/4N...	SHA-256 with MGF1 (2560bit)	3847	115782.84073 ms	30.09692 ms	31.79148 ms	1010.69826 ms²	95% with bounds [29.09231, 31.10153]	3.03521 ms	17.5282
Provable Parameters (2 Primes (1/2N...	SHA-512 with MGF1 (2560bit)	3847	115534.41875 ms	30.03234 ms	31.74501 ms	1007.74593 ms²	95% with bounds [29.02920, 31.03548]	3.03488 ms	19.5902
Provable Parameters (3 Primes (1/4N...	SHA-512 with MGF1 (2560bit)	3847	115333.49845 ms	29.98011 ms	31.62940 ms	1000.41918 ms²	95% with bounds [28.98063, 30.97960]	3.03408 ms	17.9368
Provable Parameters (2 Primes (1/2N...	SHAKE-128 (2560bit)	3847	115767.11848 ms	30.09283 ms	31.86566 ms	1015.42033 ms²	95% with bounds [29.08588, 31.09979]	3.03467 ms	17.9620
Provable Parameters (3 Primes (1/4N...	SHAKE-128 (2560bit)	3847	115933.87066 ms	30.13618 ms	31.80781 ms	1011.73649 ms²	95% with bounds [29.13105, 31.14130]	3.03429 ms	22.1624
Provable Parameters (2 Primes (1/2N...	SHAKE-256 (2560bit)	3847	116307.03245 ms	30.23318 ms	32.00628 ms	1024.40179 ms²	95% with bounds [29.22178, 31.24458]	3.03388 ms	22.3410
Provable Parameters (3 Primes (1/4N...	SHAKE-256 (2560bit)	3847	115746.46795 ms	30.08746 ms	31.66297 ms	1002.54369 ms²	95% with bounds [29.08691, 31.08801]	3.03533 ms	22.4060

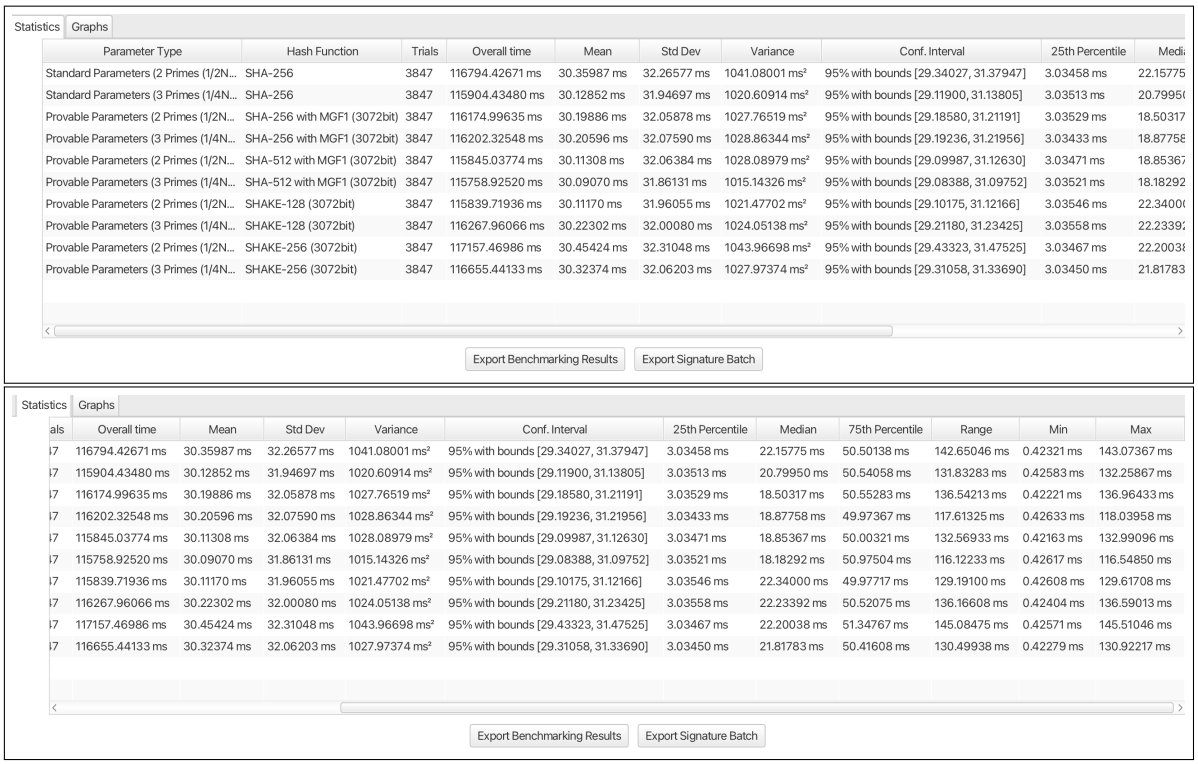
Export Benchmarking Results | Export Signature Batch

Statistics | Graphs

als	Overall time	Mean	Std Dev	Variance	Conf. Interval	25th Percentile	Median	75th Percentile	Range	Min	Max
7	115989.71794 ms	30.15069 ms	31.76533 ms	1009.03590 ms²	95% with bounds [29.14691, 31.15448]	3.03417 ms	22.40783 ms	50.84492 ms	129.18425 ms	0.42079 ms	129.60504 ms
7	115650.89943 ms	30.06262 ms	31.63348 ms	1000.67682 ms²	95% with bounds [29.06300, 31.06224]	3.03467 ms	22.40542 ms	50.31250 ms	122.30113 ms	0.42146 ms	122.72258 ms
7	116573.86325 ms	30.30254 ms	32.12846 ms	1032.23770 ms²	95% with bounds [29.28728, 31.31780]	3.03404 ms	22.36004 ms	50.47217 ms	130.90871 ms	0.42379 ms	131.33250 ms
7	115782.84073 ms	30.09692 ms	31.79148 ms	1010.69826 ms²	95% with bounds [29.09231, 31.10153]	3.03521 ms	17.52829 ms	50.54267 ms	137.77763 ms	0.42088 ms	138.19850 ms
7	115534.41875 ms	30.03234 ms	31.74501 ms	1007.74593 ms²	95% with bounds [29.02920, 31.03548]	3.03488 ms	19.59029 ms	50.11175 ms	132.19042 ms	0.42196 ms	132.61238 ms
7	115333.49845 ms	29.98011 ms	31.62940 ms	1000.41918 ms²	95% with bounds [28.98063, 30.97960]	3.03408 ms	17.93683 ms	50.60075 ms	128.66463 ms	0.42271 ms	129.08733 ms
7	115767.11848 ms	30.09283 ms	31.86566 ms	1015.42033 ms²	95% with bounds [29.08588, 31.09979]	3.03467 ms	17.96204 ms	50.62421 ms	113.48308 ms	0.42596 ms	113.90904 ms
7	115933.87066 ms	30.13618 ms	31.80781 ms	1011.73649 ms²	95% with bounds [29.13105, 31.14130]	3.03429 ms	22.16242 ms	50.28967 ms	119.85496 ms	0.42133 ms	120.27629 ms
7	116307.03245 ms	30.23318 ms	32.00628 ms	1024.40179 ms²	95% with bounds [29.22178, 31.24458]	3.03388 ms	22.34100 ms	51.16583 ms	135.87683 ms	0.42204 ms	136.29888 ms
7	115746.46795 ms	30.08746 ms	31.66297 ms	1002.54369 ms²	95% with bounds [29.08691, 31.08801]	3.03533 ms	22.40604 ms	50.66075 ms	141.37508 ms	0.42517 ms	141.80025 ms

Export Benchmarking Results | Export Signature Batch

Figure 8.14: Instantiation of PKCS#1 v1.5 with standard vs provably secure parameters (6144-bit Key Size) for signature creation



Statistics | Graphs

Parameter Type	Hash Function	Trials	Overall time	Mean	Std Dev	Variance	Conf. Interval	25th Percentile	Medi
Standard Parameters (2 Primes (1/2N...	SHA-256	3847	116794.42671 ms	30.35987 ms	32.26577 ms	1041.08001 ms²	95% with bounds [29.34027, 31.37947]	3.03458 ms	22.15775
Standard Parameters (3 Primes (1/4N...	SHA-256	3847	115904.43480 ms	30.12852 ms	31.94697 ms	1020.60914 ms²	95% with bounds [29.11900, 31.13805]	3.03513 ms	20.7995(
Provable Parameters (2 Primes (1/2N...	SHA-256 with MGF1 (3072bit)	3847	116174.99635 ms	30.19886 ms	32.05878 ms	1027.76519 ms²	95% with bounds [29.18580, 31.21191]	3.03529 ms	18.50317
Provable Parameters (3 Primes (1/4N...	SHA-256 with MGF1 (3072bit)	3847	116202.32548 ms	30.20596 ms	32.07590 ms	1028.86344 ms²	95% with bounds [29.19236, 31.21956]	3.03433 ms	18.87758
Provable Parameters (2 Primes (1/2N...	SHA-512 with MGF1 (3072bit)	3847	115845.03774 ms	30.11308 ms	32.06384 ms	1028.08979 ms²	95% with bounds [29.09987, 31.12630]	3.03471 ms	18.85367
Provable Parameters (3 Primes (1/4N...	SHA-512 with MGF1 (3072bit)	3847	115758.92520 ms	30.09070 ms	31.86131 ms	1015.14326 ms²	95% with bounds [29.08388, 31.09752]	3.03521 ms	18.18292
Provable Parameters (2 Primes (1/2N...	SHAKE-128 (3072bit)	3847	115839.71936 ms	30.11170 ms	31.96055 ms	1021.47702 ms²	95% with bounds [29.10175, 31.12166]	3.03546 ms	22.34000
Provable Parameters (3 Primes (1/4N...	SHAKE-128 (3072bit)	3847	116267.96066 ms	30.22302 ms	32.00080 ms	1024.05138 ms²	95% with bounds [29.21180, 31.23425]	3.03558 ms	22.23392
Provable Parameters (2 Primes (1/2N...	SHAKE-256 (3072bit)	3847	117157.46986 ms	30.45424 ms	32.31048 ms	1043.96698 ms²	95% with bounds [29.43323, 31.47525]	3.03467 ms	22.20038
Provable Parameters (3 Primes (1/4N...	SHAKE-256 (3072bit)	3847	116655.44133 ms	30.32374 ms	32.06203 ms	1027.97374 ms²	95% with bounds [29.31058, 31.33690]	3.03450 ms	21.81783

Export Benchmarking Results | Export Signature Batch

Statistics | Graphs

als	Overall time	Mean	Std Dev	Variance	Conf. Interval	25th Percentile	Median	75th Percentile	Range	Min	Max
7	116794.42671 ms	30.35987 ms	32.26577 ms	1041.08001 ms²	95% with bounds [29.34027, 31.37947]	3.03458 ms	22.15775 ms	50.50138 ms	142.65046 ms	0.42321 ms	143.07367 ms
7	115904.43480 ms	30.12852 ms	31.94697 ms	1020.60914 ms²	95% with bounds [29.11900, 31.13805]	3.03513 ms	20.79950 ms	50.54058 ms	131.83283 ms	0.42583 ms	132.25867 ms
7	116174.99635 ms	30.19886 ms	32.05878 ms	1027.76519 ms²	95% with bounds [29.18580, 31.21191]	3.03529 ms	18.50317 ms	50.55283 ms	136.54213 ms	0.42221 ms	136.96433 ms
7	116202.32548 ms	30.20596 ms	32.07590 ms	1028.86344 ms²	95% with bounds [29.19236, 31.21956]	3.03433 ms	18.87758 ms	49.97367 ms	117.61325 ms	0.42633 ms	118.03958 ms
7	115845.03774 ms	30.11308 ms	32.06384 ms	1028.08979 ms²	95% with bounds [29.09987, 31.12630]	3.03471 ms	18.85367 ms	50.00321 ms	132.56933 ms	0.42163 ms	132.99096 ms
7	115758.92520 ms	30.09070 ms	31.86131 ms	1015.14326 ms²	95% with bounds [29.08388, 31.09752]	3.03521 ms	18.18292 ms	50.97504 ms	116.12233 ms	0.42617 ms	116.54850 ms
7	115839.71936 ms	30.11170 ms	31.96055 ms	1021.47702 ms²	95% with bounds [29.10175, 31.12166]	3.03546 ms	22.34000 ms	49.97717 ms	129.19100 ms	0.42608 ms	129.61708 ms
7	116267.96066 ms	30.22302 ms	32.00080 ms	1024.05138 ms²	95% with bounds [29.21180, 31.23425]	3.03558 ms	22.23392 ms	50.52075 ms	136.16608 ms	0.42404 ms	136.59013 ms
7	117157.46986 ms	30.45424 ms	32.31048 ms	1043.96698 ms²	95% with bounds [29.43323, 31.47525]	3.03467 ms	22.20038 ms	51.34767 ms	145.08475 ms	0.42571 ms	145.51046 ms
7	116655.44133 ms	30.32374 ms	32.06203 ms	1027.97374 ms²	95% with bounds [29.31058, 31.33690]	3.03450 ms	21.81783 ms	50.41608 ms	130.49938 ms	0.42279 ms	130.92217 ms

Export Benchmarking Results | Export Signature Batch

Summary

Across all key sizes, PKCS#1 v1.5 benchmarking results for signature creation demonstrate subtle variations between standard and provably secure parameters. For SHA-256 with standard parameters using two primes, mean times range from 30.71927 ms to 30.35987 ms. For provably secure parameters with two primes, across different hash functions, the mean times vary from 29.96431 ms to 30.45424 ms, indicating a performance variation up to approximately 2.46%. Using three primes, standard parameters yield mean times from 30.09907 ms to 30.12852 ms, while for provably secure parameters, the range is from 30.06791 ms to 30.32374 ms, suggesting a potential variation of up to 0.65% across all key sizes.

Overall, the fluctuations in mean times across different configurations and hash functions are relatively minor. Given the scale of the operation and the small magnitude of change, these variations could likely be attributed to natural variances rather than a consistent performance trend. The standard deviations and confidence intervals reinforce that the timings are closely clustered, suggesting that any observed differences might not be statistically significant. This data indicates that using provably secure parameters for signature creation with PKCS#1 v1.5 does not introduce an overhead, and their performance is essentially on par with standard parameters.

8.4.2 Signature Creation Results (ANSI X9.31 rDSA)

Benchmarking across all key sizes for ANSI X9.31 rDSA signature creation reveals a consistent trend. With standard parameters using two primes, SHA-256 mean times range from 30.13104 ms to 30.41202 ms. For provably secure parameters with two primes, the mean times across different hash functions vary from 30.08570 ms to 30.50728 ms, demonstrating a performance

variation of up to 1.39%. With three primes, standard parameters yield mean times from 30.10680 ms to 30.41080 ms, while for provably secure parameters, the range is from 30.10680 ms to 30.86716 ms, indicating a potential variation of up to 1.49% across all key sizes.

These results align with findings from PKCS#1 v1.5, in that the difference between standard and provably secure parameters is negligible. The small fluctuations in performance times are likely inherent to computational processes and do not signify any significant statistical difference. Therefore it can be inferred that instantiation of ANSI X9.31 rDSA (for signature creation) with provably secure parameters does not introduce an overhead, and rather maintains effectiveness in line with standard parameters.

8.4.3 Signature Creation Results (ISO/IEC 9796-2:2010 Signature Scheme 1)

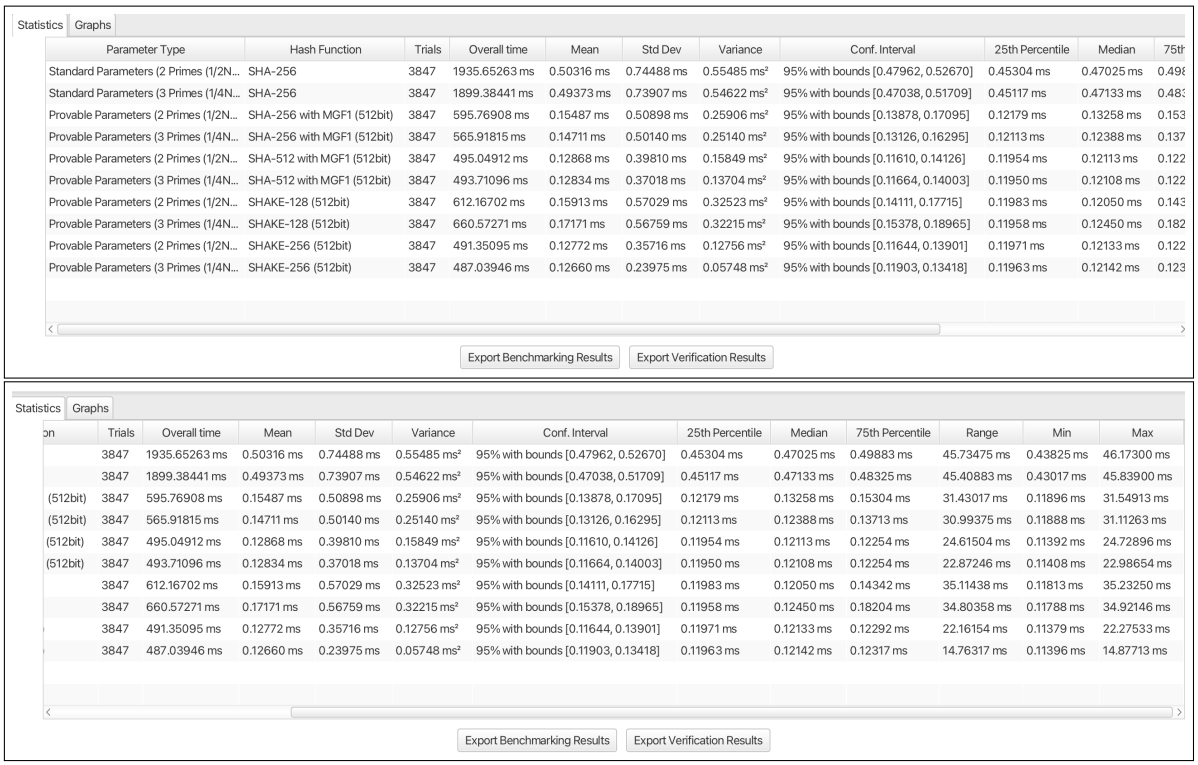
The ISO/IEC 9796-2:2010 Signature Scheme 1 exhibits a similar performance profile in its benchmarking for signature creation across all key sizes. Standard parameters using two primes for SHA-256 show mean times between 28.57240 ms and 28.72859 ms. For provably secure parameters with two primes, the mean times across different hash functions vary from 28.48672 ms to 28.74465 ms, demonstrating a performance variation of up to 1.39%. With three primes, standard parameters yield mean times from 28.60973 ms to 28.73497 ms, while for provably secure parameters, the range is from 28.54507 ms to 28.80356 ms, indicating a potential variation of up to 1.49% across all key sizes.

Like the former two schemes ISO/IEC 9796-2:2010 Signature Scheme 1 demonstrates that the use of provably secure parameters for signature creation yields only negligible differences in performance when compared to standard parameters. The small fluctuations in performance times are likely inherent to computational processes and do not signify any significant statistical difference. Therefore it can be inferred that instantiation of ISO-IEC 9796-2 Scheme 1 (for signature creation) with provably secure parameters does not introduce an overhead, and rather maintains effectiveness in line with standard parameters.

8.5 Signature Verification Results

8.5.1 Signature Verification Results (PKCS#1 v1.5)

Figure 8.15: Instantiation of PKCS#1 v1.5 with standard vs provably secure parameters (1024-bit Key Size) for signature verification



Statistics | Graphs

Parameter Type	Hash Function	Trials	Overall time	Mean	Std Dev	Variance	Conf. Interval	25th Percentile	Median	75th
Standard Parameters (2 Primes (1/2N...	SHA-256	3847	1935.65263 ms	0.50316 ms	0.74488 ms	0.55485 ms²	95% with bounds [0.47962, 0.52670]	0.45304 ms	0.47025 ms	0.49
Standard Parameters (3 Primes (1/4N...	SHA-256	3847	1899.38441 ms	0.49373 ms	0.73907 ms	0.54622 ms²	95% with bounds [0.47038, 0.51709]	0.45117 ms	0.47133 ms	0.48
Provable Parameters (2 Primes (1/2N...	SHA-256 with MGF1 (512bit)	3847	595.76908 ms	0.15487 ms	0.50898 ms	0.25906 ms²	95% with bounds [0.13878, 0.17095]	0.12179 ms	0.13258 ms	0.153
Provable Parameters (3 Primes (1/4N...	SHA-256 with MGF1 (512bit)	3847	565.91815 ms	0.14711 ms	0.50140 ms	0.25140 ms²	95% with bounds [0.13126, 0.16295]	0.12113 ms	0.12388 ms	0.137
Provable Parameters (2 Primes (1/2N...	SHA-512 with MGF1 (512bit)	3847	495.04912 ms	0.12868 ms	0.39810 ms	0.15849 ms²	95% with bounds [0.11610, 0.14126]	0.11954 ms	0.12113 ms	0.122
Provable Parameters (3 Primes (1/4N...	SHA-512 with MGF1 (512bit)	3847	493.71096 ms	0.12834 ms	0.37018 ms	0.13704 ms²	95% with bounds [0.11664, 0.14003]	0.11950 ms	0.12108 ms	0.122
Provable Parameters (2 Primes (1/2N...	SHAKE-128 (512bit)	3847	612.16702 ms	0.15913 ms	0.57029 ms	0.32523 ms²	95% with bounds [0.14111, 0.17715]	0.11983 ms	0.12050 ms	0.143
Provable Parameters (3 Primes (1/4N...	SHAKE-128 (512bit)	3847	660.57271 ms	0.17171 ms	0.56759 ms	0.32215 ms²	95% with bounds [0.15378, 0.18965]	0.11958 ms	0.12450 ms	0.182
Provable Parameters (2 Primes (1/2N...	SHAKE-256 (512bit)	3847	491.35095 ms	0.12772 ms	0.35716 ms	0.12756 ms²	95% with bounds [0.11644, 0.13901]	0.11971 ms	0.12133 ms	0.122
Provable Parameters (3 Primes (1/4N...	SHAKE-256 (512bit)	3847	487.03946 ms	0.12660 ms	0.23975 ms	0.05748 ms²	95% with bounds [0.11903, 0.13418]	0.11963 ms	0.12142 ms	0.123

Export Benchmarking Results | Export Verification Results

Statistics | Graphs

on	Trials	Overall time	Mean	Std Dev	Variance	Conf. Interval	25th Percentile	Median	75th Percentile	Range	Min	Max
	3847	1935.65263 ms	0.50316 ms	0.74488 ms	0.55485 ms²	95% with bounds [0.47962, 0.52670]	0.45304 ms	0.47025 ms	0.49883 ms	45.73475 ms	0.43825 ms	46.17300 ms
	3847	1899.38441 ms	0.49373 ms	0.73907 ms	0.54622 ms²	95% with bounds [0.47038, 0.51709]	0.45117 ms	0.47133 ms	0.48325 ms	45.40883 ms	0.43017 ms	45.83900 ms
(512bit)	3847	595.76908 ms	0.15487 ms	0.50898 ms	0.25906 ms²	95% with bounds [0.13878, 0.17095]	0.12179 ms	0.13258 ms	0.15304 ms	31.43017 ms	0.11896 ms	31.54913 ms
(512bit)	3847	565.91815 ms	0.14711 ms	0.50140 ms	0.25140 ms²	95% with bounds [0.13126, 0.16295]	0.12113 ms	0.12388 ms	0.13713 ms	30.99375 ms	0.11888 ms	31.11263 ms
(512bit)	3847	495.04912 ms	0.12868 ms	0.39810 ms	0.15849 ms²	95% with bounds [0.11610, 0.14126]	0.11954 ms	0.12113 ms	0.12254 ms	24.61504 ms	0.11392 ms	24.72896 ms
(512bit)	3847	493.71096 ms	0.12834 ms	0.37018 ms	0.13704 ms²	95% with bounds [0.11664, 0.14003]	0.11950 ms	0.12108 ms	0.12254 ms	22.87246 ms	0.11408 ms	22.98654 ms
	3847	612.16702 ms	0.15913 ms	0.57029 ms	0.32523 ms²	95% with bounds [0.14111, 0.17715]	0.11983 ms	0.12050 ms	0.14342 ms	35.11438 ms	0.11813 ms	35.23250 ms
	3847	660.57271 ms	0.17171 ms	0.56759 ms	0.32215 ms²	95% with bounds [0.15378, 0.18965]	0.11958 ms	0.12450 ms	0.18204 ms	34.80358 ms	0.11788 ms	34.92146 ms
	3847	491.35095 ms	0.12772 ms	0.35716 ms	0.12756 ms²	95% with bounds [0.11644, 0.13901]	0.11971 ms	0.12133 ms	0.12292 ms	22.16154 ms	0.11379 ms	22.27533 ms
	3847	487.03946 ms	0.12660 ms	0.23975 ms	0.05748 ms²	95% with bounds [0.11903, 0.13418]	0.11963 ms	0.12142 ms	0.12317 ms	14.76317 ms	0.11396 ms	14.87713 ms

Export Benchmarking Results | Export Verification Results

Figure 8.16: Instantiation of PKCS#1 v1.5 with standard vs provably secure parameters (2048-bit Key Size) for signature verification

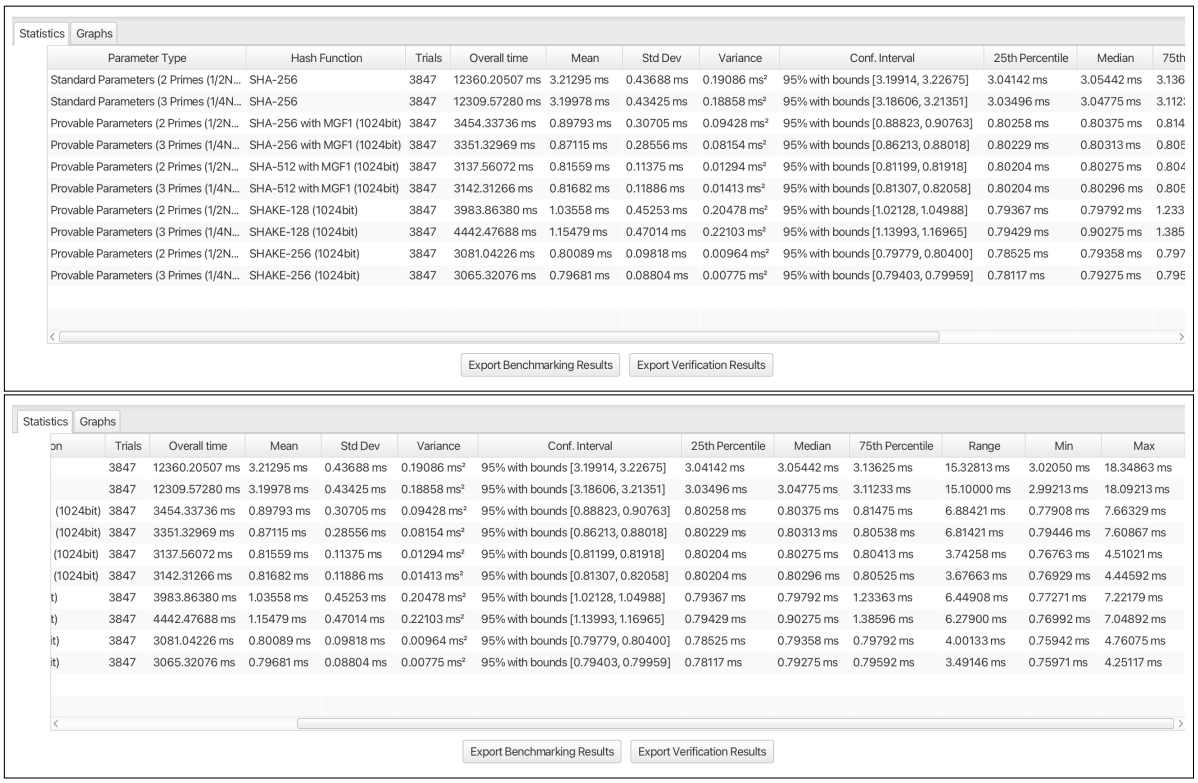


Figure 8.17: Instantiation of PKCS#1 v1.5 with standard vs provably secure parameters (3072-bit Key Size) for signature verification

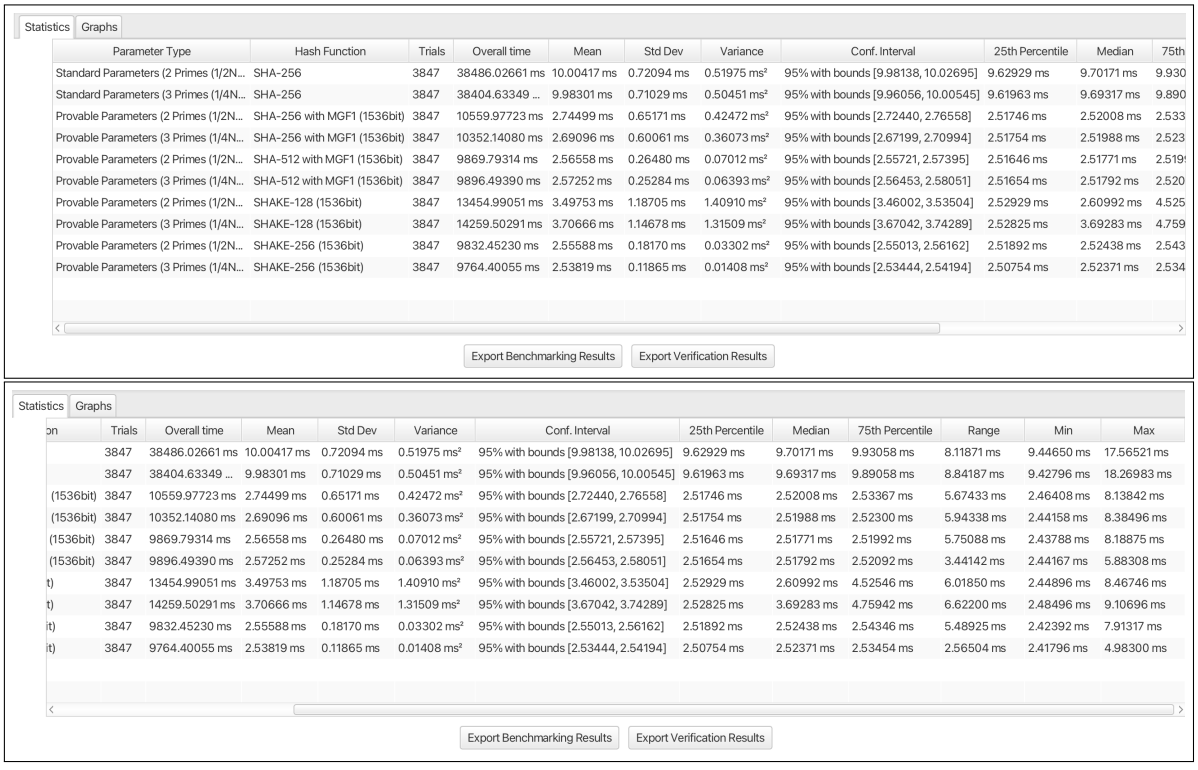
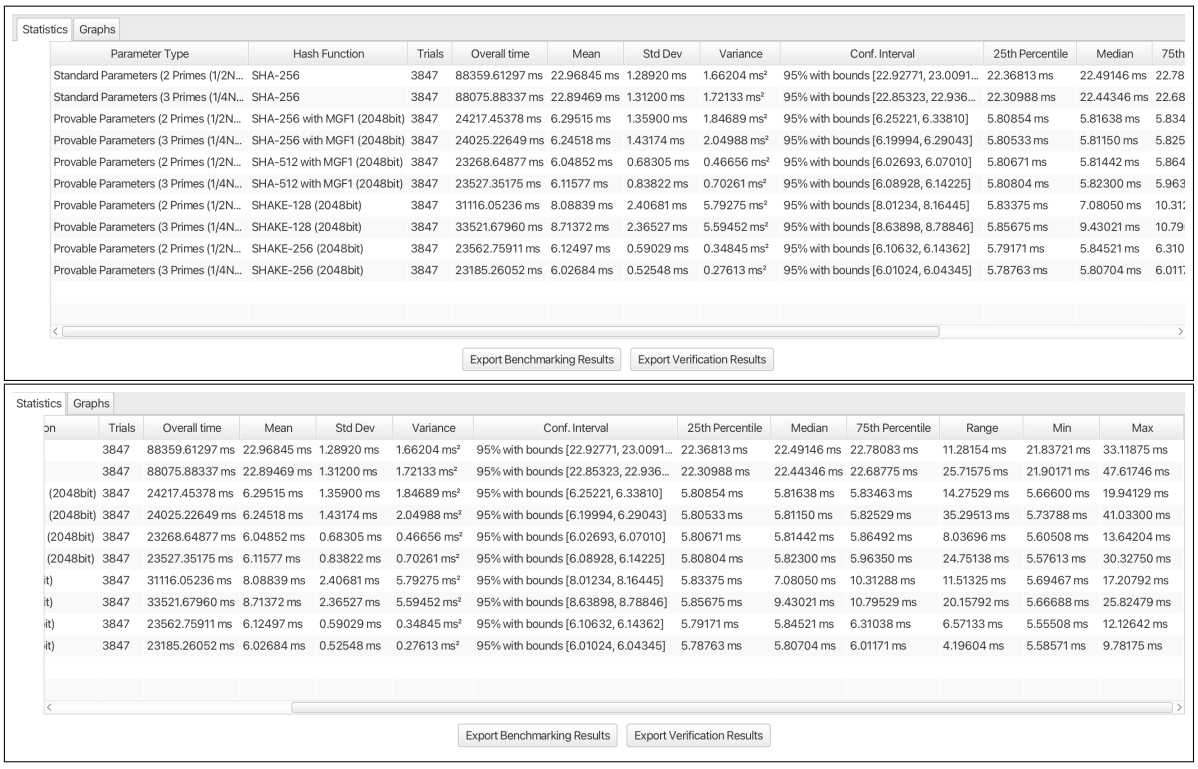


Figure 8.18: Instantiation of PKCS#1 v1.5 with standard vs provably secure parameters (4096-bit Key Size) for signature verification



Statistics | Graphs

Parameter Type	Hash Function	Trials	Overall time	Mean	Std Dev	Variance	Conf. Interval	25th Percentile	Median	75th
Standard Parameters (2 Primes (1/2N...	SHA-256	3847	88359.61297 ms	22.96845 ms	1.28920 ms	1.66204 ms²	95% with bounds [22.92771, 23.0091...	22.36813 ms	22.49146 ms	22.78
Standard Parameters (3 Primes (1/4N...	SHA-256	3847	88075.88337 ms	22.89469 ms	1.31200 ms	1.72133 ms²	95% with bounds [22.85323, 22.936...	22.30988 ms	22.44346 ms	22.68
Provable Parameters (2 Primes (1/2N...	SHA-256 with MGF1 (2048bit)	3847	24217.45378 ms	6.29515 ms	1.35900 ms	1.84689 ms²	95% with bounds [6.25221, 6.33810]	5.80854 ms	5.81638 ms	5.834
Provable Parameters (3 Primes (1/4N...	SHA-256 with MGF1 (2048bit)	3847	24025.22649 ms	6.24518 ms	1.43174 ms	2.04988 ms²	95% with bounds [6.19994, 6.29043]	5.80533 ms	5.81150 ms	5.825
Provable Parameters (2 Primes (1/2N...	SHA-512 with MGF1 (2048bit)	3847	23268.64877 ms	6.04852 ms	0.68305 ms	0.46656 ms²	95% with bounds [6.02693, 6.07010]	5.80671 ms	5.81442 ms	5.864
Provable Parameters (3 Primes (1/4N...	SHA-512 with MGF1 (2048bit)	3847	23527.35175 ms	6.11577 ms	0.83822 ms	0.70261 ms²	95% with bounds [6.08928, 6.14225]	5.80804 ms	5.82300 ms	5.963
Provable Parameters (2 Primes (1/2N...	SHAKE-128 (2048bit)	3847	31116.05236 ms	8.08839 ms	2.40681 ms	5.79275 ms²	95% with bounds [8.01234, 8.16445]	5.83375 ms	7.08050 ms	10.31
Provable Parameters (3 Primes (1/4N...	SHAKE-128 (2048bit)	3847	33521.67960 ms	8.71372 ms	2.36527 ms	5.59452 ms²	95% with bounds [8.63898, 8.78846]	5.85675 ms	9.43021 ms	10.79
Provable Parameters (2 Primes (1/2N...	SHAKE-256 (2048bit)	3847	23562.75911 ms	6.12497 ms	0.59029 ms	0.34845 ms²	95% with bounds [6.10632, 6.14362]	5.79171 ms	5.84521 ms	6.310
Provable Parameters (3 Primes (1/4N...	SHAKE-256 (2048bit)	3847	23185.26052 ms	6.02684 ms	0.52548 ms	0.27613 ms²	95% with bounds [6.01024, 6.04345]	5.78763 ms	5.80704 ms	6.011

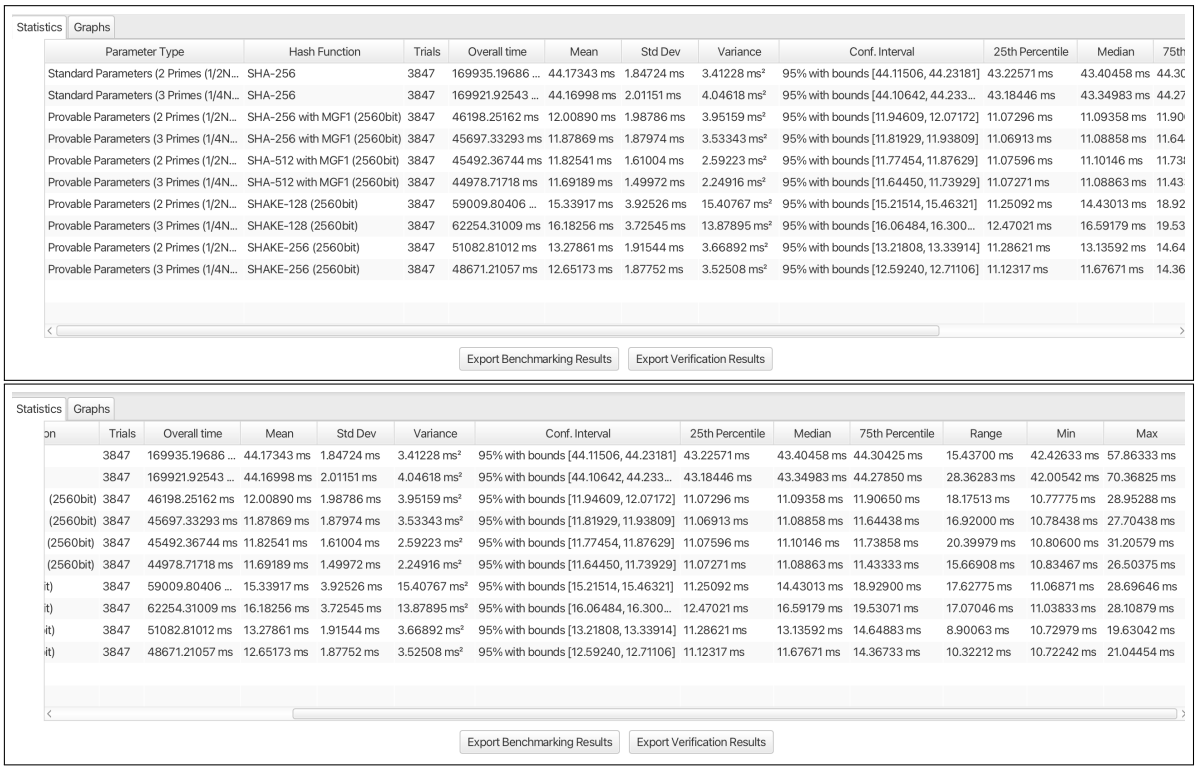
Export Benchmarking Results | Export Verification Results

Statistics | Graphs

on	Trials	Overall time	Mean	Std Dev	Variance	Conf. Interval	25th Percentile	Median	75th Percentile	Range	Min	Max
	3847	88359.61297 ms	22.96845 ms	1.28920 ms	1.66204 ms²	95% with bounds [22.92771, 23.0091...	22.36813 ms	22.49146 ms	22.78083 ms	11.28154 ms	21.83721 ms	33.11875 ms
	3847	88075.88337 ms	22.89469 ms	1.31200 ms	1.72133 ms²	95% with bounds [22.85323, 22.936...	22.30988 ms	22.44346 ms	22.68775 ms	25.71575 ms	21.90171 ms	47.61746 ms
(2048bit)	3847	24217.45378 ms	6.29515 ms	1.35900 ms	1.84689 ms²	95% with bounds [6.25221, 6.33810]	5.80854 ms	5.81638 ms	5.83463 ms	14.27529 ms	5.66600 ms	19.94129 ms
(2048bit)	3847	24025.22649 ms	6.24518 ms	1.43174 ms	2.04988 ms²	95% with bounds [6.19994, 6.29043]	5.80533 ms	5.81150 ms	5.82529 ms	35.29513 ms	5.73788 ms	41.03300 ms
(2048bit)	3847	23268.64877 ms	6.04852 ms	0.68305 ms	0.46656 ms²	95% with bounds [6.02693, 6.07010]	5.80671 ms	5.81442 ms	5.86492 ms	8.03696 ms	5.60508 ms	13.64204 ms
(2048bit)	3847	23527.35175 ms	6.11577 ms	0.83822 ms	0.70261 ms²	95% with bounds [6.08928, 6.14225]	5.80804 ms	5.82300 ms	5.96350 ms	24.75138 ms	5.57613 ms	30.32750 ms
it)	3847	31116.05236 ms	8.08839 ms	2.40681 ms	5.79275 ms²	95% with bounds [8.01234, 8.16445]	5.83375 ms	7.08050 ms	10.31288 ms	11.51325 ms	5.69467 ms	17.20792 ms
it)	3847	33521.67960 ms	8.71372 ms	2.36527 ms	5.59452 ms²	95% with bounds [8.63898, 8.78846]	5.85675 ms	9.43021 ms	10.79529 ms	20.15792 ms	5.66688 ms	25.82479 ms
it)	3847	23562.75911 ms	6.12497 ms	0.59029 ms	0.34845 ms²	95% with bounds [6.10632, 6.14362]	5.79171 ms	5.84521 ms	6.31038 ms	6.57133 ms	5.55508 ms	12.12642 ms
it)	3847	23185.26052 ms	6.02684 ms	0.52548 ms	0.27613 ms²	95% with bounds [6.01024, 6.04345]	5.78763 ms	5.80704 ms	6.01171 ms	4.19604 ms	5.58571 ms	9.78175 ms

Export Benchmarking Results | Export Verification Results

Figure 8.19: Instantiation of PKCS#1 v1.5 with standard vs provably secure parameters (5120-bit Key Size) for signature verification



Statistics | Graphs

Parameter Type	Hash Function	Trials	Overall time	Mean	Std Dev	Variance	Conf. Interval	25th Percentile	Median	75th
Standard Parameters (2 Primes (1/2N...	SHA-256	3847	169935.19686 ...	44.17343 ms	1.84724 ms	3.41228 ms²	95% with bounds [44.11506, 44.23181]	43.22571 ms	43.40458 ms	44.30
Standard Parameters (3 Primes (1/4N...	SHA-256	3847	169921.92543 ...	44.16998 ms	2.01151 ms	4.04618 ms²	95% with bounds [44.10642, 44.233...	43.18446 ms	43.34983 ms	44.27
Provable Parameters (2 Primes (1/2N...	SHA-256 with MGF1 (2560bit)	3847	46198.25162 ms	12.00890 ms	1.98786 ms	3.95159 ms²	95% with bounds [11.94609, 12.07172]	11.07296 ms	11.09358 ms	11.90
Provable Parameters (3 Primes (1/4N...	SHA-256 with MGF1 (2560bit)	3847	45697.33293 ms	11.87869 ms	1.87974 ms	3.53343 ms²	95% with bounds [11.81929, 11.93809]	11.06913 ms	11.08858 ms	11.64
Provable Parameters (2 Primes (1/2N...	SHA-512 with MGF1 (2560bit)	3847	45492.36744 ms	11.82541 ms	1.61004 ms	2.59223 ms²	95% with bounds [11.77454, 11.87629]	11.07596 ms	11.10146 ms	11.73
Provable Parameters (3 Primes (1/4N...	SHA-512 with MGF1 (2560bit)	3847	44978.71718 ms	11.69189 ms	1.49972 ms	2.24916 ms²	95% with bounds [11.64450, 11.73929]	11.07271 ms	11.08863 ms	11.43
Provable Parameters (2 Primes (1/2N...	SHAKE-128 (2560bit)	3847	59009.80406 ...	15.33917 ms	3.92526 ms	15.40767 ms²	95% with bounds [15.21514, 15.46321]	11.25092 ms	14.43013 ms	18.92
Provable Parameters (3 Primes (1/4N...	SHAKE-128 (2560bit)	3847	62254.31009 ms	16.18256 ms	3.72545 ms	13.87895 ms²	95% with bounds [16.06484, 16.300...	12.47021 ms	16.59179 ms	19.53
Provable Parameters (2 Primes (1/2N...	SHAKE-256 (2560bit)	3847	51082.81012 ms	13.27861 ms	1.91544 ms	3.66892 ms²	95% with bounds [13.21808, 13.33914]	11.28621 ms	13.13592 ms	14.64
Provable Parameters (3 Primes (1/4N...	SHAKE-256 (2560bit)	3847	48671.21057 ms	12.65173 ms	1.87752 ms	3.52508 ms²	95% with bounds [12.59240, 12.71106]	11.12317 ms	11.67671 ms	14.36

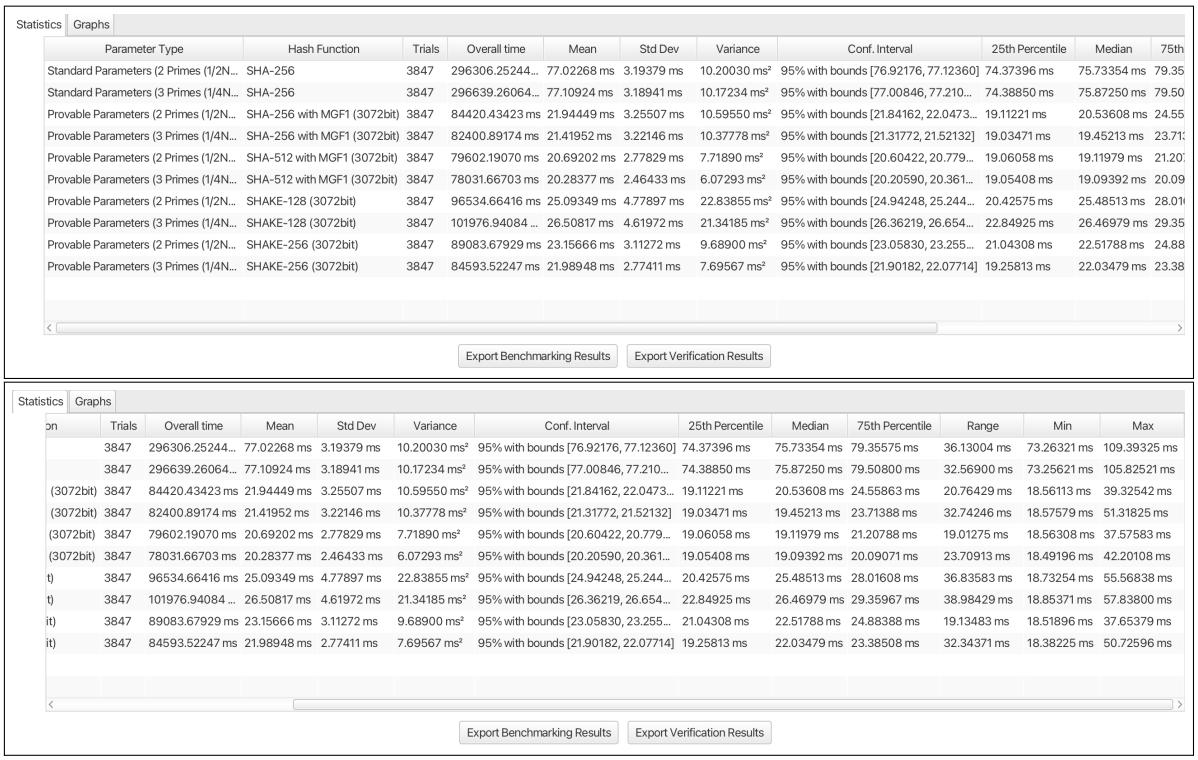
Export Benchmarking Results | Export Verification Results

Statistics | Graphs

on	Trials	Overall time	Mean	Std Dev	Variance	Conf. Interval	25th Percentile	Median	75th Percentile	Range	Min	Max
	3847	169935.19686 ...	44.17343 ms	1.84724 ms	3.41228 ms²	95% with bounds [44.11506, 44.23181]	43.22571 ms	43.40458 ms	44.30425 ms	15.43700 ms	42.42633 ms	57.86333 ms
	3847	169921.92543 ...	44.16998 ms	2.01151 ms	4.04618 ms²	95% with bounds [44.10642, 44.233...	43.18446 ms	43.34983 ms	44.27850 ms	28.36283 ms	42.00542 ms	70.36825 ms
(2560bit)	3847	46198.25162 ms	12.00890 ms	1.98786 ms	3.95159 ms²	95% with bounds [11.94609, 12.07172]	11.07296 ms	11.09358 ms	11.90650 ms	18.17513 ms	10.77775 ms	28.95288 ms
(2560bit)	3847	45697.33293 ms	11.87869 ms	1.87974 ms	3.53343 ms²	95% with bounds [11.81929, 11.93809]	11.06913 ms	11.08858 ms	11.64438 ms	16.92000 ms	10.78438 ms	27.70438 ms
(2560bit)	3847	45492.36744 ms	11.82541 ms	1.61004 ms	2.59223 ms²	95% with bounds [11.77454, 11.87629]	11.07596 ms	11.10146 ms	11.73858 ms	20.39979 ms	10.80600 ms	31.20579 ms
(2560bit)	3847	44978.71718 ms	11.69189 ms	1.49972 ms	2.24916 ms²	95% with bounds [11.64450, 11.73929]	11.07271 ms	11.08863 ms	11.43333 ms	15.66908 ms	10.83467 ms	26.50375 ms
it)	3847	59009.80406 ...	15.33917 ms	3.92526 ms	15.40767 ms²	95% with bounds [15.21514, 15.46321]	11.25092 ms	14.43013 ms	18.92900 ms	17.62775 ms	11.06871 ms	28.69646 ms
it)	3847	62254.31009 ms	16.18256 ms	3.72545 ms	13.87895 ms²	95% with bounds [16.06484, 16.300...	12.47021 ms	16.59179 ms	19.53071 ms	17.07046 ms	11.03833 ms	28.10879 ms
it)	3847	51082.81012 ms	13.27861 ms	1.91544 ms	3.66892 ms²	95% with bounds [13.21808, 13.33914]	11.28621 ms	13.13592 ms	14.64883 ms	8.90063 ms	10.72979 ms	19.63042 ms
it)	3847	48671.21057 ms	12.65173 ms	1.87752 ms	3.52508 ms²	95% with bounds [12.59240, 12.71106]	11.12317 ms	11.67671 ms	14.36733 ms	10.32212 ms	10.72242 ms	21.04454 ms

Export Benchmarking Results | Export Verification Results

Figure 8.20: Instantiation of PKCS#1 v1.5 with standard vs provably secure parameters (6144-bit Key Size) for signature verification



Statistics | Graphs

Parameter Type	Hash Function	Trials	Overall time	Mean	Std Dev	Variance	Conf. Interval	25th Percentile	Median	75th
Standard Parameters (2 Primes (1/2N...	SHA-256	3847	296306.25244...	77.02268 ms	3.19379 ms	10.20030 ms²	95% with bounds [76.92176, 77.12360]	74.37396 ms	75.73354 ms	79.35
Standard Parameters (3 Primes (1/4N...	SHA-256	3847	296639.26064...	77.10924 ms	3.18941 ms	10.17234 ms²	95% with bounds [77.00846, 77.210...	74.38850 ms	75.87250 ms	79.50
Provable Parameters (2 Primes (1/2N...	SHA-256 with MGF1 (3072bit)	3847	84420.43423 ms	21.94449 ms	3.25507 ms	10.59550 ms²	95% with bounds [21.84162, 22.0473...	19.11221 ms	20.53608 ms	24.55
Provable Parameters (3 Primes (1/4N...	SHA-256 with MGF1 (3072bit)	3847	82400.89174 ms	21.41952 ms	3.22146 ms	10.37778 ms²	95% with bounds [21.31772, 21.52132]	19.03471 ms	19.45213 ms	23.71
Provable Parameters (2 Primes (1/2N...	SHA-512 with MGF1 (3072bit)	3847	79602.19070 ms	20.69202 ms	2.77829 ms	7.71890 ms²	95% with bounds [20.60422, 20.779...	19.06058 ms	19.11979 ms	21.20
Provable Parameters (3 Primes (1/4N...	SHA-512 with MGF1 (3072bit)	3847	78031.66703 ms	20.28377 ms	2.46433 ms	6.07293 ms²	95% with bounds [20.20590, 20.361...	19.05408 ms	19.09392 ms	20.09
Provable Parameters (2 Primes (1/2N...	SHAKE-128 (3072bit)	3847	96534.66416 ms	25.09349 ms	4.77897 ms	22.83855 ms²	95% with bounds [24.94248, 25.244...	20.42575 ms	25.48513 ms	28.01
Provable Parameters (3 Primes (1/4N...	SHAKE-128 (3072bit)	3847	101976.94084 ...	26.50817 ms	4.61972 ms	21.34185 ms²	95% with bounds [26.36219, 26.654...	22.84925 ms	26.46979 ms	29.35
Provable Parameters (2 Primes (1/2N...	SHAKE-256 (3072bit)	3847	89083.67929 ms	23.15666 ms	3.11272 ms	9.68900 ms²	95% with bounds [23.05830, 23.255...	21.04308 ms	22.51788 ms	24.88
Provable Parameters (3 Primes (1/4N...	SHAKE-256 (3072bit)	3847	84593.52247 ms	21.98948 ms	2.77411 ms	7.69567 ms²	95% with bounds [21.90182, 22.07714]	19.25813 ms	22.03479 ms	23.38

Export Benchmarking Results | Export Verification Results

Statistics | Graphs

on	Trials	Overall time	Mean	Std Dev	Variance	Conf. Interval	25th Percentile	Median	75th Percentile	Range	Min	Max
	3847	296306.25244...	77.02268 ms	3.19379 ms	10.20030 ms²	95% with bounds [76.92176, 77.12360]	74.37396 ms	75.73354 ms	79.35575 ms	36.13004 ms	73.26321 ms	109.39325 ms
	3847	296639.26064...	77.10924 ms	3.18941 ms	10.17234 ms²	95% with bounds [77.00846, 77.210...	74.38850 ms	75.87250 ms	79.50800 ms	32.56900 ms	73.25621 ms	105.82521 ms
(3072bit)	3847	84420.43423 ms	21.94449 ms	3.25507 ms	10.59550 ms²	95% with bounds [21.84162, 22.0473...	19.11221 ms	20.53608 ms	24.55863 ms	20.76429 ms	18.56113 ms	39.32542 ms
(3072bit)	3847	82400.89174 ms	21.41952 ms	3.22146 ms	10.37778 ms²	95% with bounds [21.31772, 21.52132]	19.03471 ms	19.45213 ms	23.71388 ms	32.74246 ms	18.57579 ms	51.31825 ms
(3072bit)	3847	79602.19070 ms	20.69202 ms	2.77829 ms	7.71890 ms²	95% with bounds [20.60422, 20.779...	19.06058 ms	19.11979 ms	21.20788 ms	19.01275 ms	18.56308 ms	37.57583 ms
(3072bit)	3847	78031.66703 ms	20.28377 ms	2.46433 ms	6.07293 ms²	95% with bounds [20.20590, 20.361...	19.05408 ms	19.09392 ms	20.09071 ms	23.70913 ms	18.49196 ms	42.20108 ms
t)	3847	96534.66416 ms	25.09349 ms	4.77897 ms	22.83855 ms²	95% with bounds [24.94248, 25.244...	20.42575 ms	25.48513 ms	28.01608 ms	36.83583 ms	18.73254 ms	55.56838 ms
t)	3847	101976.94084 ...	26.50817 ms	4.61972 ms	21.34185 ms²	95% with bounds [26.36219, 26.654...	22.84925 ms	26.46979 ms	29.35967 ms	38.98429 ms	18.85371 ms	57.83800 ms
it)	3847	89083.67929 ms	23.15666 ms	3.11272 ms	9.68900 ms²	95% with bounds [23.05830, 23.255...	21.04308 ms	22.51788 ms	24.88388 ms	19.13483 ms	18.51896 ms	37.65379 ms
it)	3847	84593.52247 ms	21.98948 ms	2.77411 ms	7.69567 ms²	95% with bounds [21.90182, 22.07714]	19.25813 ms	22.03479 ms	23.38508 ms	32.34371 ms	18.38225 ms	50.72596 ms

Export Benchmarking Results | Export Verification Results

Standard Parameters

- 2 Primes: Mean verification time with SHA-256 ranges from 0.50316 ms (1024-bit) to 77.02268 ms (6144-bit).
- 3 Primes: Mean verification time with SHA-256 ranges from 0.49373 ms (1024-bit) to 77.10924 ms (6144-bit).

Provable Parameters (2 Primes)

- SHA-512 with MGF1: Shows improvements in verification time compared to standard parameter instantiations with 2 Primes, with reductions ranging from 73.13% to 74.62%. This hash function also consistently exhibits lower variance compared to other provably secure parameter instantiations
- SHAKE-256: Demonstrates improvements in verification time compared to standard parameters with 2 Primes, with reductions ranging from 69.94% to 75.07%. It generally shows lower variance across the various key sizes.
- SHA-256 with MGF1: Demonstrates improvement in verification time compared to standard parameters with 2 Primes with reductions ranging from 69.22% to 72.81%. While it shows moderately low variance, it is generally higher compared to SHA-512 with MGF1 and SHAKE-256.
- SHAKE-128: Offers smallest in verification time compared to standard parameters with 2 Primes from 64.78% to 68.37%. However, it consistently presents the highest variance among the provably secure parameter instantiations.

Provable Parameters (3 Primes)

- SHA-512 with MGF1: Provides improvements in verification time compared to standard parameters with 3 Primes, with reductions ranging from 73.29% to 74.47%. Similar to its performance with 2 Primes, it exhibits low variance across key sizes.
- SHAKE-256: Shows improvements in verification time compared to standard parameters with 3 Primes, with reductions ranging from 71.36% to 75.10%. It consistently demonstrates one of the lowest variances.
- SHA-256 with MGF1: Shows improvement in verification time compared to standard parameters instantiations with 3 Primes, ranging from 70.20% to 73.11%. While its variance is moderately low, it is higher compared to SHA-512 with MGF1 and SHAKE-256.
- SHAKE-128: Provides the smallest improvements in verification time compared to standard parameters with 3 Primes, ranging from 61.94% to 65.62%, and consistently shows the highest variance. Notably, when evaluated outside the 3 Prime context, SHAKE-128 shows relatively smaller improvements in verification time compared to its performance with 2 Primes across all key sizes.

Summary

The analysis revealed a surprising outcome: employing provably secure parameters in the PKCS#1 v1.5 signature scheme led to significant improvement in verification times as compared to doing the same with standard parameters, challenging initial expectations. This unexpected efficiency gain can be attributed primarily to the relationship between the public exponent e and the hash function's output size in determining the computational efficiency of signature verification. It's established that smaller values of e result in more efficient verification processes, as they require fewer bits and operations for exponentiation. The results were aligned with this.

The key determinant of computational time was found to be the use of smaller 'e' values in provably secure configurations, as opposed to the arbitrary values in standard configurations. The role of the hash function and its output size, while still relevant, was secondary to the choice of 'e'. This effectively balanced any potential negative impact on computational time from employing large hash output sizes ($\lambda/2$) in provably secure instantiations of the PKCS#1 scheme.

Upon isolating the value of e, variations in signature verification time emerged across the range of hash functions used in provably secure instantiations. This was evident even without altering the hash output size, but merely by employing different variable-length hash functions with their varying levels of security and/or construction mechanisms used to achieve the same output size.

For instance, the results revealed a direct correlation between the strength of the hash function and the relative improvements in verification times. Higher-strength hash functions such as SHA-512 with MGF1 and SHAKE-256 not only demonstrated the greatest improvements in verification times but also maintained the lowest variances across key sizes. On the other hand, SHAKE-128, which represents the lowest security level among the variable-length hash functions, consistently demonstrated the highest variance and the least improvement among the provably secure instantiations for all key sizes.

SHAKE-128 in different prime settings

For all hash functions used in provably secure instantiations bar SHAKE-128, the improvement in verification times as compared between 2 prime and 3 primes was negligible. However, Interestingly for SHAKE-128, there was a noticeable difference between 2 and 3 prime provably secure instantiations. 3 prime provably secure instantiations were slower in signature verifications and thus lead to smallest improvement in verification times for every key size. The difference in improvement between 2 and 3 primes for SHAKE-128 ranged from 1.79% to 3.18% less improvement for 3 primes compared to 2 primes.

Overall the data indicates there is a significant improvement in verification time from instantiating the PKCS#1 v1.5 signature scheme using provably secure parameters, with improvements for the full set of key sizes ranging from 64.78% to 75.07% in the 2 prime setting and 61.94% to 75.1% in the 3 prime setting, contingent upon the security level of the hash function employed for instantiation. Additionally the data indicates that while SHAKE-128 offers improvement in verification times in both 2 and 3 prime settings, the relative efficiency is somewhat reduced in the 3 prime setup for all key sizes.

8.5.2 Signature Verification Results (ANSI X9.31 rDSA)

The analysis of the ANSI X9.31 rDSA scheme corroborates the findings from the PKCS#1 v1.5 scheme, indicating that provably secure parameters substantially enhance verification times across all key sizes.

Hash Function Influence:

The Results from ANSI X9.31 rDSA scheme reinforce the secondary impact of the hash function's output size and type on computational time in relation to values of e. Moreover, higher-strength hash functions, demonstrated superior efficiency in verification time reduction with low variance.

SHAKE-128 Discrepancy:

SHAKE-128, within provably secure instantiations of ANSI X9.31 rDSA, exhibited notable differences between 2-prime and 3-prime settings. The improvement in verification time was less pronounced in 3-prime configurations, with differences ranging from 0.95% to 8.6%. Notably, the discrepancy is more pronounced than that observed in the PKCS#1 v1.5 scheme.

Overall, the analysis of the ANSI X9.31 rDSA scheme demonstrates that employing provably secure parameters leads to significant improvements in verification times, with reductions ranging from 70.1% to 75.57% in 2-prime settings and 61.83% to 75.07% in 3-prime settings. These gains are influenced by the security level of the hash function employed, with SHAKE-128 presenting unique variations in efficiency between different prime settings.

8.5.3 Signature Verification Results (ISO/IEC 9796-2:2010 Signature Scheme 1)

The analysis of the ISO/IEC 9796-2:2010 Signature Scheme 1 corroborates the findings from the latter two schemes indicating that provably secure parameters substantially enhance verification times across all key sizes.

Hash Function Influence:

The Results from ISO/IEC 9796-2:2010 Signature Scheme 1 again reinforce the secondary impact of the hash function's output size and type on computational time in relation to values of e. Moreover, higher-strength hash functions, demonstrated superior efficiency in verification time reduction with low variance.

SHAKE-128 Discrepancy:

SHAKE-128, within provably secure instantiations of ISO/IEC 9796-2:2010 Signature Scheme 1 exhibited notable differences between 2-prime and 3-prime settings. Again the improvement in verification time was less pronounced in 3-prime configurations, with differences ranging from 0.23% to 10.6%.

Overall, the analysis of the ISO/IEC 9796-2:2010 Signature Scheme 1 scheme demonstrates that employing provably secure parameters leads to significant improvements in verification times, with reductions ranging from 69.64% to 75.26% in 2-prime settings and 62.14% to 75.08% in 3-prime settings. These gains are influenced by the security level of the hash function employed, with SHAKE-128 presenting unique variations in efficiency between different prime settings.

Chapter 9: **Conclusion**

To be done.

Bibliography

[1] B. Kaliski, *PKCS #1: RSA Encryption Version 1.5*, RFC 2313, Mar. 1998. DOI: 10.17487/RFC2313. [Online]. Available: https://www.rfc-editor.org/info/rfc2313.

[2] J. Schaad, B. Kaliski, and R. Housley, "Additional algorithms and identifiers for rsa cryptography for use in the internet x. 509 public key infrastructure certificate and certificate revocation list (crl) profile," Tech. Rep., 2005.

[3] T. Jager, S. A. Kakvi, and A. May, "On the security of the pkcs# 1 v1. 5 signature scheme," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1195–1208.

[4] M. Bellare and P. Rogaway, "The exact security of digital signatures-how to sign with rsa and rabin," in *International conference on the theory and applications of cryptographic techniques*, Springer, 1996, pp. 399–416.

[5] J. Jonsson, "Security proofs for the rsa-pss signature scheme and its variants," *Cryptology ePrint Archive*, 2001.

[6] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch, *PKCS #1: RSA Cryptography Specifications Version 2.2*, RFC 8017, Nov. 2016. DOI: 10.17487/RFC8017. [Online]. Available: https://www.rfc-editor.org/info/rfc8017.

[7] J. Jonsson and B. Kaliski, *Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1*, RFC 3447, Feb. 2003. DOI: 10.17487/RFC3447. [Online]. Available: https://www.rfc-editor.org/info/rfc3447.

[8] J.-S. Coron, "Security proof for partial-domain hash signature schemes," in *Advances in Cryptology—CRYPTO 2002: 22nd Annual International Cryptology Conference Santa Barbara, California, USA, August 18–22, 2002 Proceedings 22*, Springer, 2002, pp. 613–626.

[9] D. Bleichenbacher, "Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs# 1," in *Advances in Cryptology—CRYPTO'98: 18th Annual International Cryptology Conference Santa Barbara, California, USA August 23–27, 1998 Proceedings 18*, Springer, 1998, pp. 1–12.

[10] H. Finney, "Bleichenbacher's rsa signature forgery based on implementation error," *http://www. imc. org/ietf-openpgp/mail-archive/msg14307. html*, 2006.

[11] S. A. Kakvi and E. Kiltz, "Optimal security proofs for full domain hash, revisited," *Journal of Cryptology*, vol. 31, pp. 276–306, 2018.

[12] G. Ateniese, B. Magri, and D. Venturi, "Subversion-resilient signature schemes," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 364–375.

[13] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978, ISSN: 0001-0782. DOI: 10.1145/359340.359342. [Online]. Available: https://doi.org/10.1145/359340.359342.

[14] ANSI, *ANSI X9.31:1998: Digital Signatures Using Reversible Public Key Cryptography for the Financial Services Industry (RDSA)*. Washington, DC, USA: Accredited Standards Committee/X9, Sep. 1998, p. 66. [Online]. Available: https://global.ihs.com/doc_detail.cfm?item_s_key=00326003&item_key_date=011231&rid=GS.

[15] ISO/IEC, *ISO/IEC 9796-2:2010: Information technology – Security techniques – Digital signature schemes giving message recovery – Part 2: Integer factorization based mechanisms*. Geneva, CH: International Organization for Standardization/International Electrotechnical Commission, Dec. 2010, p. 54. [Online]. Available: https://www.iso.org/standard/54788.html.

[16] J.-S. Coron, D. Naccache, and J. P. Stern, "On the security of rsa padding," in *Advances in Cryptology — CRYPTO' 99*, M. Wiener, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 1–18.

[17] J.-S. Coron, D. Naccache, M. Tibouchi, and R.-P. Weinmann, "Practical cryptanalysis of iso 9796-2 and emv signatures," *Journal of Cryptology*, vol. 29, pp. 632–656, 2016.

[18] D. Coppersmith, M. Franklin, J. Patarin, and M. Reiter, "Low-exponent rsa with related messages," in *International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 1996, pp. 1–9.

[19] J.-S. Coron, M. Joye, D. Naccache, and P. Paillier, "New attacks on pkcs# 1 v1. 5 encryption," in *International conference on the theory and applications of cryptographic techniques*, Springer, 2000, pp. 369–381.

[20] V. Klíma, O. Pokorný, and T. Rosa, "Attacking rsa-based sessions in ssl/tls," in *Cryptographic Hardware and Embedded Systems - CHES 2003*, C. D. Walter, Ç. K. Koç, and C. Paar, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 426–440.

[21] J. P. Degabriele, A. Lehmann, K. G. Paterson, N. P. Smart, and M. Strefler, "On the joint security of encryption and signature in emv," in *Topics in Cryptology–CT-RSA 2012: The Cryptographers' Track at the RSA Conference 2012, San Francisco, CA, USA, February 27–March 2, 2012. Proceedings*, Springer, 2012, pp. 116–135.

[22] R. Bardou, R. Focardi, Y. Kawamoto, L. Simionato, G. Steel, and J.-K. Tsay, "Efficient padding oracle attacks on cryptographic hardware," in *Annual Cryptology Conference*, Springer, 2012, pp. 608–625.

[23] C. Meyer, J. Somorovsky, E. Weiss, J. Schwenk, S. Schinzel, and E. Tews, "Revisiting {ssl/tls} implementations: New bleichenbacher side channels and attacks," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 733–748.

[24] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in paas clouds," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 990–1003.

[25] T. Jager, J. Schwenk, and J. Somorovsky, "On the security of tls 1.3 and quic against weaknesses in pkcs# 1 v1. 5 encryption," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 1185–1196.

[26] T. Jager, J. Schwenk, and J. Somorovsky, "Practical invalid curve attacks on tls-ecdh," in *Computer Security–ESORICS 2015: 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part I 20*, Springer, 2015, pp. 407–425.

[27] H. Böck, J. Somorovsky, and C. Young, "Return of {bleichenbacher's} oracle threat ({{{{{robot}}}}}}," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 817–849.

[28] U. Kühn, A. Pyshkin, E. Tews, and R.-P. Weinmann, "Variants of bleichenbacher's low-exponent attack on pkcs# 1 rsa signatures," *SICHERHEIT 2008–Sicherheit, Schutz und Zuverlässigkeit. Beiträge der 4. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik eV (GI)*, 2008.

[29] MITRE Corporation. "Cve-2006-4339." (2006), [Online]. Available: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-4339`.

[30] M. Bellare and P. Rogaway, "Random oracles are practical: A paradigm for designing efficient protocols," in *Proceedings of the 1st ACM Conference on Computer and Communications Security*, ser. CCS '93, Fairfax, Virginia, USA: Association for Computing Machinery, 1993, pp. 62–73, ISBN: 0897916298. DOI: `10.1145/168588.168596`. [Online]. Available: `https://doi.org/10.1145/168588.168596`.

[31] M. Bellare and S. Micali, "How to sign given any trapdoor permutation," *J. ACM*, vol. 39, no. 1, pp. 214–233, Jan. 1992, ISSN: 0004-5411. DOI: 10.1145/147508.147537. [Online]. Available: https://doi.org/10.1145/147508.147537.

[32] M. Bellare and M. Yung, "Certifying cryptographic tools: The case of trapdoor permutations," in *Advances in Cryptology — CRYPTO' 92*, E. F. Brickell, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 442–460.

[33] M. Bellare and M. Yung, "Certifying permutations: Noninteractive zero-knowledge based on any trapdoor permutation," *Journal of Cryptology*, vol. 9, pp. 149–166, 1996.

[34] S. A. Kakvi, E. Kiltz, and A. May, "Certifying rsa," in *Advances in Cryptology – ASIACRYPT 2012*, X. Wang and K. Sako, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 404–414.

[35] C. Peikert and B. Waters, "Lossy trapdoor functions and their applications," in *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, ser. STOC '08, Victoria, British Columbia, Canada: Association for Computing Machinery, 2008, pp. 187–196, ISBN: 9781605580470. DOI: 10.1145/1374376.1374406. [Online]. Available: https://doi.org/10.1145/1374376.1374406.

[36] J.-S. Coron, "Optimal security proofs for pss and other signature schemes," in *Advances in Cryptology — EUROCRYPT 2002*, L. R. Knudsen, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 272–287.

[37] C. Cachin, S. Micali, and M. Stadler, "Computationally private information retrieval with polylogarithmic communication," in *Advances in Cryptology — EUROCRYPT '99*, J. Stern, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 402–414.

[38] A. Lysyanskaya, S. Micali, L. Reyzin, and H. Shacham, "Sequential aggregate signatures from trapdoor permutations," in *Advances in Cryptology - EUROCRYPT 2004*, C. Cachin and J. L. Camenisch, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 74–90.

[39] D. Coppersmith, "Small solutions to polynomial equations, and low exponent rsa vulnerabilities," *Journal of cryptology*, vol. 10, no. 4, pp. 233–260, 1997.

[40] M. Dworkin, *Sha-3 standard: Permutation-based hash and extendable-output functions*, en, 2015-08-04 2015. DOI: https://doi.org/10.6028/NIST.FIPS.202.

[41] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, *The keccak sha-3 submission*, Submission to NIST (Round 3), 2011. [Online]. Available: https://keccak.team/files/Keccak-reference-3.0.pdf.

[42] S. A. Kakvi, "Sok: Comparison of the security of real world rsa hash-and-sign signatures," in *Security Standardisation Research*, T. van der Merwe, C. Mitchell, and M. Mehrnezhad, Eds., Cham: Springer International Publishing, 2020, pp. 91–113, ISBN: 978-3-030-64357-7.

[43] ISO/IEC. "What is cryptography?" (2024), [Online]. Available: https://www.iso.org/information-security/what-is-cryptography#toc5.

[44] J. P. Degabriele and K. G. Paterson, "Attacking the ipsec standards in encryption-only configurations," in *2007 IEEE Symposium on Security and Privacy (SP '07)*, 2007, pp. 335–349. DOI: 10.1109/SP.2007.8.

[45] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, "Why eve and mallory love android: An analysis of android ssl (in)security," ser. CCS '12, Raleigh, North Carolina, USA: Association for Computing Machinery, 2012, pp. 50–61, ISBN: 9781450316514. DOI: 10.1145/2382196.2382205. [Online]. Available: https://doi.org/10.1145/2382196.2382205.

[46] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: Validating ssl certificates in non-browser software," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12, Raleigh, North Carolina, USA: Association for Computing Machinery, 2012, pp. 38–49, ISBN: 9781450316514. DOI: 10.1145/2382196.2382204. [Online]. Available: https://doi.org/10.1145/2382196.2382204.

[47] British Standards Institution (BSI), *BSI (British Standards Institution) Website*, https://bsol-bsigroup-com.ezproxy01.rhul.ac.uk/.

[48] Oxford University Press, *The Oxford 3000*, https://www.oxfordlearnersdictionaries.com/about/oxford3000, 2022.

[49] GitHub User: sapbmw, *The Oxford 3000 on GitHub*, https://github.com/sapbmw/The-Oxford-3000, 2022.