An overview of secure multiparty computation and its application to digital asset custody

Matt O'Grady

Technical Report

RHUL–ISG–2022–8

11 April 2022

Information Security Group
Royal Holloway University of London
Egham, Surrey, TW20 0EX
United Kingdom

2110808

Royal Holloway
University of London

An Overview of Secure Multiparty Computation and its Application to Digital Asset Custody

MSc Project Report

Overleaf Word Count: 18,086

September 1, 2021

Executive Summary

Secure multiparty computation (MPC) as a subfield of cryptography has existed for just under 40 years. In essence, a secure multiparty computation protocol is executed by two or more parties who wish to jointly compute the output of an arbitrary function, without sacrificing the privacy of their respective inputs. Initially, MPC was branded as a purely theoretical concept, however since the late 2000s, it has been used to solve a myriad of real-world problems. As the adoption of MPC continues to increases within everyday applications, it is now essential for information security practitioners to be aware of the fundamental concepts underpinning its functionality.

In the first half of this report, we aim to provide a thorough overview of secure multiparty computation, ranging from the mathematical primitives that are commonly used to construct MPC protocols, to the current and future applications of MPC as a whole. With a view to appeal to an audience with a wide range of mathematical abilities, we have included detailed examples and explanations throughout.

In the second half of this report, we apply our newly attained knowledge to a particular application of MPC, namely threshold signature schemes, and their potential application within digital asset self-custody solutions. Our main focus in relation to this is a state-of-the-art threshold Schnorr signature scheme known as FROST, which we deconstruct with the aim of not only providing a detailed summary of its operation, but further how its goals are achieved through the use of secure multiparty computation. FROST is a relatively new threshold signature scheme, having only been introduced by Komlo et al. in December 2020. As such, it is yet to be implemented and utilised for any real-world applications. Therefore, as a part of this report we have also produced a proof-of-concept Python implementation and demonstration of FROST. Finally, we compare FROST to three other mechanisms, including a multi-signature scheme known as MuSig2, with a view to consider the benefits and drawbacks of utilising each in the context of digital asset self-custody.

Contents

1 Introduction

First concretely defined by Yao [1] in 1982, secure multiparty computation (MPC) is a subfield of cryptography that aims to solve the following problem statement:

Assume that m participants, $P_1, \ldots, P_m$, each possess a private value $x_1, \ldots, x_m$ respectively and wish to jointly compute the result of an arbitrary function, $f(x_1, \ldots, x_m)$. Is this possible to achieve, without a mutually trusted intermediary and without the need for any participant to ever reveal the value of their individual input to the other participants?

This seemingly abstract concept is often illustrated in the literature with the so-called millionaires' problem, which was first stated and solved by Yao in [1]. The millionaires' problem can be seen as a special case of MPC involving only two participants, each of whom are millionaires', that wish to determine who is the richest without either millionaire needing to disclosing their net-worth the other and without outsourcing the computation to a third party. As time has gone on, the defined goals of MPC have evolved to not only include the assurance of privacy for participants inputs, but also four other main objectives, as defined by Lindell and Pinkas [2]:

- Correctness – the result of the joint computation cannot be altered by any adversary that wishes to subvert it.
- Guaranteed Output Delivery – an adversary must not be able to prevent participants from receiving the result of the joint computation.
- Independence of inputs – participants must not be able to construct their input to be dependent on the input of any other participants.
- Fairness – in the presence of an adversary, either all or none of the participants receive the result of the joint computation.

Secure multiparty computation is a powerful concept in the sense that it has been shown by Goldreich et al. [3] that *any* arbitrary function can be securely computed, however progress towards the application of MPC with the aim of solving real-world problems stalled for many decades due to the inefficiency of the solutions that were initially proposed.

1.1 Motivation & Objectives

It was not until the turn of the 21st century that MPC matured to become an efficient tool that could be utilised for practical purposes. Indeed, in November

2019, a consortium of businesses known as the MPC Alliance was formed, with the intention of raising awareness and adoption of MPC across various industries, and saw a 3x increase in membership over its first year of operation [4]. Given this, it is clear there is now a greater interest in MPC and its possible applications than we have ever previously seen, and as a result, academic literature on the subject has grown substantially. However, much of this literature is either overly mathematical in nature or excessively abstract due to a lack of mathematical detail. In both cases, this hinders the comprehension of the average reader. As such, the first half of this report seeks to find a balance between the two approaches commonly found in the literature, acting as a well-rounded overview of MPC that includes both a technical review of the mathematical techniques that are used to achieve the objectives of MPC, but also a summary of the notable real-world applications that utilise them. To supplement this, we will include concrete examples where appropriate, to illustrate that the mathematical techniques do indeed achieve their intended goals. Again, this is an aspect that is not frequently included in the literature.

In essence, our intention in the first half of the report is to provide the reader with an appreciation of the big picture in relation to the underlying mathematical construction of MPC and how MPC in general can be applied to a wide variety of real-world problems. In contrast, the second half of this report delves deeper into a specific application of MPC, namely threshold signature schemes. Our aim in this section is to first introduce the reader to threshold signatures, how they differ from traditional signature schemes and to describe the operation of a commercial implementation of a threshold signature scheme that is used to facilitate the custody of digital assets (in particular, Bitcoin). We then aim to explore a state-of-the-art threshold variant of the Schnorr signature scheme known as FROST, with a view to explain in detail how it achieves the objectives of secure multiparty computation – an aspect not considered in the original paper [5]. To reinforce the reader's understanding of FROST, we also aim to produce a proof-of-concept Python implementation of the scheme and a demonstration of its use. Our final objective is to consider the potential application of FROST to Bitcoin transactions and custody, with a view to analyse and discuss alternative mechanisms with similar properties.

1.2 Structure

This report is comprised of four further sections. In section two, we will begin by introducing the reader to the core concepts and fundamental mathematical primitives required to achieve MPC, including oblivious transfer, garbled circuits and three forms of secret sharing. In section three, we then move on to describe, at a relatively high level, two current commerical applications of MPC in the realm of privacy preserving auctions and privacy preserving analytics. Following this, we briefly present three potential future applications of MPC currently being investigated within academia. Section four – the last section within the main body of this report – takes a deep-dive into threshold signatures, with emphasis on the FROST signature scheme, including a proof-of-concept Python implementation and a discussion of alternative mechanisms. Finally, we make some concluding remarks in section five.

2 Preliminaries

Secure computation is a broad subfield of cryptography, involving many unique concepts and mathematical tools. As such, this section is designed to familiarise the reader with the most fundamental concepts in secure computation, including the models used describe potential attacks against an MPC protocol, the common mathematical techniques used to achieve the goals of MPC.

2.1 Security Models

Similar to any other cryptographic protocol, an MPC protocol must incorporate robust countermeasures to ensure that any entity that wishes to subvert the goals of the protocol (as described in Section 1) does not succeed. What differentiates MPC protocols from other cryptographic protocols, however, is that entities that participate must be equally wary of the ulterior motives possessed by other authorised participants, not just adversaries external to the protocol execution that possess malicious intentions. MPC participants with these characteristics are commonly known as corrupted parties, each of which are categorised widely in the literature [6, 7] by their propensity to subvert the protocol:

- A *semi-honest* (or, *honest-but-curious*) participant follows the protocol exactly as specified, without deviation. However, they will attempt to analyse the messages exchanged between themselves and other parties with a view to subvert its goals, such as the privacy of other participants inputs. An MPC protocol is said to offer *passive* security if it can withstand attack from semi-honest participants.

- A *malicious* participant, on the other hand, uses any means available to them to to subvert the protocol. This includes deviating from the protocol specification in an arbitrary fashion, to the extent that the corrupt party may simply choose to cease communication during execution. An MPC protocol is said to offer *active* security if it can withstand attack from malicious participants.

- Finally, for sake of completeness, an *honest* party is a non-corrupt party. In other words, an honest party is a party that wishes to participate in the protocol, without any desire to do subvert the goals of the protocol.

2.2 Mathematical Primitives

2.2.1 Oblivious Transfer

Oblivious transfer is a cryptographic protocol that describes the transmission of information between two parties, a sender and recipient, whilst ensuring the following properties of the exchange are satisfied:

- Assuming the sender is in possession of two distinct pieces of information, x_1 and x_2, the recipient must only obtain a single piece of information, x_i, from the sender. In particular, this x_i must be nothing but the information that was requested by the recipient.
- The sender cannot be certain which piece of information has been obtained by the recipient during the transfer. In other words, the sender remains oblivious to which x_i they have sent to the recipient.

This definition differs somewhat to the definition of oblivious transfer first proposed by Michael O. Rabin in [8]. In his paper, Rabin describes a protocol that involves the transfer of a single of piece of information, which the sender knowns has been requested by the recipient, and is designed to ensure that the recipient either receives the requested information, or nothing at all, both with a probability of 1/2. In Rabin's protocol, the sender has no way of knowing the outcome of this transfer – i.e., the sender does not know whether the recipient obtained the requested information or not. Instead, our definition follows what is known as a 1-out-of-2 (or 1–2) oblivious transfer protocol, that was first defined by Even et al. in [9]. This protocol was then generalised by Brassard et al. in [10] to what is known as a 1-out-of-n oblivious transfer, however, in this report we will only summarise 1-out-of-2 oblivious transfer.

To execute a 1-out-of-2 oblivious transfer protocol, as defined by Even et al. two parties P_1 and P_2, must first agree which underlying public-key cryptosystem to utilise. The protocol defined in [9] is written in generic notation to allow the participants to use, in theory, *any* public-key cryptosystem (PKCS) of their choice. Therefore, assuming that P_1 and P_2 have agreed to utilise the RSA cryptosystem (which is in fact the suggested PKCS in [9]), P_1 can execute an oblivious transfer to P_2 involving two units of information, x_1 and x_2, as follows:

1. P_1 generates a fresh RSA modulus n, encryption exponent e and decryption exponent d. As per textbook RSA, P_1 must keep d private and send their public key (n, e) to P_2.
2. P_1 randomly generates and sends P_2 two values, $y_0 \neq y_1$, such that:

$$y_0, y_1 \in \{0, 1, 2, \dots, n-1\}.$$

3. P_2 picks a value $c \in \{0, 1\}$, representing the unit of information x_c they wish to request from P_1. Then, P_2 must generate a random $k \in \{0, 1, 2, \ldots, n-1\}$ and send the following to P_1:

$$z \equiv y_c + k^e \pmod{n}$$

4. P_1 must generate and send two values, $x'_0 \equiv x_0 + k_0 \pmod{n}$ and $x'_1 \equiv x_1 + k_1 \pmod{n}$ to P_2, where:

$$\begin{aligned} k_0 &= (z - y_0)^d \\ k_1 &= (z - y_1)^d. \end{aligned}$$

5. P_2 can obtain their requested information by calculating $x_c \equiv x'_c - k \pmod{n}$.

We can show that the two required properties from the definition of oblivious transfer are achieved. First, P_2 is able to obtain their requested information because:

$$\begin{aligned} x'_c - k &= x_c + k_c - k \\ &= x_c + (z - y_c)^d - k \\ &= x_c + (y_c + k^e - y_c)^d - k \\ &= x_c + k^{ed} - k \\ &= x_c + k - k \\ &= x_c. \end{aligned}$$

However, P_2 is not able to reveal the information they did not request (i.e., x_{1-c}). This can be shown by attempting the same method as above with the aim of obtaining x_{1-c}:

$$\begin{aligned} x'_{1-c} - k &= x_{1-c} + k_{1-c} - k \\ &= x_{1-c} + (z - y_{1-c})^d - k \\ &= x_{1-c} + (y_c + k^e - y_{1-c})^d - k \\ &\neq x_{1-c}. \end{aligned}$$

The inequality holds because y_0 and y_1 are generated by P_1 such that $y_0 \neq y_1$.

Furthermore, P_1 is unable to determine with certainty which piece of information, x_c, was requested by P_2. This is because the random value y_c that corresponds to the requested information, x_c, is concealed by introducing the k^e modulo n term, for a secret value, k, generated by P_2. This is best illustrated with an example – assume P_1 defines an RSA public key, $(n, e) = (55, 3)$. Then, assume that P_1 sets $y_0 = 7$ and $y_1 = 12$, and recieves $z = 4$ from P_2. If P_1 believes that P_2 has chosen $c = 0$, P_1 must solve the following for k:

$$7 + k^3 \equiv 4 \pmod{55}$$

This yields $k = 13 \pmod{55}$. However, P_1 could alternatively believe that P_2 has chosen $c = 1$. Applying the same logic as above, this would mean that $k \equiv 53 \pmod{55}$. Therefore, there are two values of $k \pmod{55}$ for which $z = 4$, meaning both possible values of c have equal probability of being the actual value chosen by P_2. As such, if P_2 never reveals k, the value of c cannot be determined with certainty by P_1. At first, oblivious transfer may appear to bear no relation to secure multiparty computation, however this will become clear in the next subsection.

2.2.2 Garbled Circuits

The concept of garbled circuits and their application to secure two-party computation is attributed to Andrew Yao, who first discussed the idea in an oral presentation of one of his papers [11]. Although Goldreich et al. were the first to publish a written description of the concept in [3], Goldreich himself has since given credit to Yao in [12] for first describing what he calls *scrambled circuits* – an alternate name for garbled circuits. The more common name for the technique, garbled circuits, was first coined by Beaver et al. in [13].

As primitives of secure computation, garbled circuits and oblivious transfer differ somewhat. Neither an oblivious transfer protocol nor a garbled circuit can alone be used to facilitate secure computation. Instead, Yao's garbled circuit protocol facilitates secure two-party computation of a predefined function, but the efficacy of such a protocol requires on an underlying oblivious transfer protocol. We will now describe a simplified version of Yao's garbled circuit protocol that can achieve passively secure two-party secure computation, based on recent work by Lindell and Pinkas [14], which is cited as the first formal proof of security of Yao's protocol. An example function will be included in this description, to aid in reader comprehension, and the description will also be followed by an explanation as to how this protocol achieves the goal of input privacy as required by secure multiparty computation.

Assume two parties, P_1 and P_2, both possess a private input, x and y respectively, and wish to jointly compute the output of the function $z = f(x, y)$ without revealing their private inputs to each other at any time during the computation. Yao's garbled circuit protocol is able to facilitate this in the following set of steps:

1. P_1 must first construct a boolean circuit representation of the function, f. A boolean (or binary) circuit is simply a representation of the function f using logic gates, connected by wires that carry the input and output bits of the function. The logic gates that could comprise a boolean circuit include – but are not limited to – AND ($\wedge$), OR ($\vee$) and XOR ($\oplus$) gates and their logical compliments. This is in contrast to an arithmetic circuit, which is also comprised of gates and wires, however in an arithmetic circuit these gates perform arithmetic operations (e.g., addition and multiplication), and the wires carry arbitrary field elements as inputs and outputs, instead of bits. For sake of simplicity, we will assume that f can be described by a single XOR gate – that is, $f(x, y) = x \oplus y$, where both x and y represent a single bit input from P_1 and P_2 respectively. However in reality, many more interconnected gates would be required to represent a complex function as a boolean circuit. A boolean circuit can, in turn, be represented in the form of a circuit diagram containing logic gates, each of which accept two inputs and one output, known as wires. Figure 2.1 shows the circuit diagram for $f(x, y)$, where w_x and w_y represent the wires that will be used to accept the input to the XOR gate for P_1 and P_2 respectively. Further, w_z represents the output wire of the XOR gate.

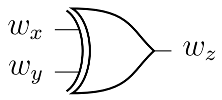

Figure 2.1: XOR Gate.

To evaluate a boolean circuit, a truth table can be constructed, as seen in Table 2.1. This table represents all possible input combinations across both input wires, w_x and w_y, and the resultant output on w_z.

w_x	w_y	w_z
0	0	0
0	1	1
1	0	1
1	1	0

Table 2.1: Truth table representation of $f(x, y) = x \oplus y = w_1 \oplus w_2$.

2. P_1 must then *garble* the boolean representation of f that was produced in the previous step. To garble a circuit, P_1 must first generate two random cryptographic keys for each wire present in the boolean circuit. In our case, the

	w_x	
Value	0	1
Label	k_0^x	k_1^x

	w_y	
Value	0	1
Label	k_0^y	k_1^y

	w_z	
Value	0	1
Label	k_0^z	k_1^z

Table 2.2: Wire value to label / key correspondence.

boolean circuit for f contains only three wires, therefore six cryptographic keys must be generated. These keys must be compatible with a symmetric key encryption algorithm, E, that was agreed between P_1 and P_2 before a run of the protocol. The choice of E is important, but will be discussed after this protocol description. Each wire is then assigned two keys (often known as labels), one which will represent the value 0, and one to represent the value 1, as seen in Figure 2.2. For example, k_0^x and k_1^x represent the keys generated to represent the bits 0 and 1 respectively, when sent over input wire w_x. A similar value-to-label correspondence is also performed for w_y and w_z.

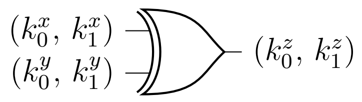

Figure 2.2: XOR Gate using labels.

This value to key correspondence is stored and kept secret by P_1 in a form similar to what can be seen in Table 2.2. P_1 must then replace the values in Table 2.1 using this value-to-label correspondence, the result of which can be seen in Table 2.3.

w_x	w_y	w_z
k_0^x	k_0^y	k_0^z
k_0^x	k_1^y	k_1^z
k_1^x	k_0^y	k_1^z
k_1^x	k_1^y	k_0^z

Table 2.3: Truth table representation of $f(x, y)$ using labels.

In this form, the truth table can now be garbled by P_1. To do so, P_1 must perform a double-encryption on each of the four possible output labels for w_z. This process requires two encryption keys, which can be retrieved from each input wire column of the corresponding row in the truth table. The output of this process can be seen in Table 2.4.

w_x	w_y	w_z	$E_{w_z}(E_{w_y}(w^z))$
k_0^x	k_0^y	k_0^z	$E_{k_0^x}(E_{k_0^y}(k_0^z))$
k_0^x	k_1^y	k_1^z	$E_{k_0^x}(E_{k_1^y}(k_1^z))$
k_1^x	k_0^y	k_1^z	$E_{k_1^x}(E_{k_0^y}(k_1^z))$
k_1^x	k_1^y	k_0^z	$E_{k_1^x}(E_{k_1^y}(k_0^z))$

Table 2.4: Garbled truth table representation of $f(x, y)$.

The final step of the garbling process requires P_1 to randomly permute the encrypted column of the garbled truth table. As a result, P_1 will be left with a table similar (but not necessarily an exact match) to Table 2.5. This is performed to ensure that when this table is received by P_2, they will be unable to determine the real output value by simply comparing rows in the garbled truth table to a standard XOR truth table, such as Table 2.1.

$E_{w_z}(E_{w_y}(w^z))$
$E_{k_0^x}(E_{k_1^y}(k_1^z))$
$E_{k_1^x}(E_{k_0^y}(k_1^z))$
$E_{k_0^x}(E_{k_0^y}(k_0^z))$
$E_{k_1^x}(E_{k_1^y}(k_0^z))$

Table 2.5: Garbled truth table representation of the output of $f(x, y)$.

P_1 must then send Table 2.5 to P_1 in preparation for the computation phase. Again, if $f(x, y)$ was a more complex function, P_1 would have to send a garbled truth table for each gate in the circuit. Further, P_1 must also send the labels that represent their choice of input to the function. This is either one of two values, k_0^x or k_1^x that will be sent over w_x. In our example, we assume that P_1 chooses k_0^x.

3. Once P_2 has received the garbled output seen in Table 2.5 and P_1's choice of input, P_2 must choose an input for wire w_y. However, because P_1 has kept the value-to-label correspondence secret, P_2 does not know what label corresponds to their desired input. This is where 1-out-of-2 oblivious transfer is required, as described in Section 2.2.1. Assuming that P_2 chooses the input value of 1, they will therefore obtain the label k_1^y from P_1 via oblivious transfer.

4. Now that P_2 is in possession of two labels, each representing a participants private input, they can evaluate the output of the function. To do so, P_2 must refer to Table 2.5 that was received in the previous step, and decrypt the value that has been double-encrypted with the labels in their possession. In our case, P_2 is in possession of k_0^x and k_1^y, therefore P_2 is able to decrypt

$E_{k_0^x}(E_{k_1^y}(k_1^z))$ and obtain k_1^z. Again, P_1 is the only party in possession of the value-to-label correspondence in Table 2.2, therefore P_2 must inform P_1 that the output of the evaluation is k_1^z, then P_1 will be able to map this label back to an output value using Table 2.2 and reveal this to P_2.

It is not necessarily clear why Yao's protocol ensures that both parties inputs remain private from each other during this computation, therefore we will now explain how this is achieved from the perspective of both participants:

- The crucial piece of information required to be kept secret to preserve the privacy of the input of P_1 is the value-to-label correspondence in Table 2.2. These labels are randomly chosen by P_1, therefore when P_1 sends their choice to P_2 (in our example, this is k_0^x), P_2 has no method to determine which real input value k_0^x corresponds to without knowledge of Table 2.2.

- The input privacy of P_2 is achieved using oblivious transfer. As described in Section 2.2.1, oblivious transfer allows P_2 to retrieve the label corresponding to their desired input, without P_1 learning which choice was taken and further P_2 is not able to learn the label corresponding to the other choice of input, which would allow them to decrypt more than one ciphertext from Table 2.5.

Finally, we mentioned in step two of the protocol that the choice of encryption algorithm was important. In step four of the protocol, P_2 is presented with Table 2.5, which from their perspective is simply a table containing four ciphertexts. Therefore, even if P_2 uses the two label / keys to decrypt all ciphertexts, only one resultant plaintext represents the output of the function. The other three plaintext labels will be meaningless as the correct keys will not have be used during the decryption, and P_1 will be unable to verify which label is the correct output without access to the value-to-label correspondence in Table 2.2. Therefore, [14] suggests that the encryption algorithm should have an efficiently verifiable range, meaning P_1 is able to efficiently verify which ciphertext is the result of a given key, and therefore P_1 will be able to determine which decryption corresponds to the correct output label of the function. However, as [14] notes, ensuring correctness of output can also be achieved by other means, such as explicit but randomly permuted indices, such as those used in [15].

2.2.3 Secret Sharing

The last primitive we discuss is *secret sharing*, first formalised by Shamir [16], but also discovered concurrently by Blakley [17]. Broadly, this term and is used to describe the act of distributing a secret value, S, between n individual parties, $P_1, \ldots, P_n$, such that each party cannot recover S without knowledge of the values (or *shares*) possessed by the other participants.

2.2.3.1 Additive Secret Sharing

Although never explicitly defined in either [16] or [17], addition over a finite field can be used as a rudimentary secret sharing scheme.

To construct an additive secret sharing scheme, a finite field must first be chosen. This is commonly taken to be the field of integers modulo a prime p, which we will denote $\mathbb{F}_p$. Now, say we wish to share a secret, S, between n parties, $P_1, \dots, P_n$, the following set of steps must be performed:

1. Generate a random set of shares, $\{s_1, \dots, s_{n-1}\}$, such that $s_i \in \mathbb{F}_p$.
2. Set $s_n = S - \sum_{i=1}^{n-1} s_i$.
3. Distribute each share, s_i, to the its respective owner, P_i.

As a result, each party has now has a share of the secret, S. To recover S, all parties must all disclose their respective shares s_i to one another and subsequently calculate $S = \sum_{i=1}^{n} s_i$. To illustrate this scheme, consider the following example, where we share the secret $S = 41$ between $n = 3$ parties, having chosen the finite field $\mathbb{F}_p$, where $p = 47$ is prime. Then, we can share S as follows:

1. Generate $\{12, 25\}$ as a set of random values in $\mathbb{F}_{47}$. As such, $s_1 = 12$ and $s_2 = 25$.
2. Let $s_3 = 41 - 12 - 25 \equiv 4 \pmod{47}$
3. Distribute $s_1 = 12$, $s_2 = 25$ and $s_3 = 4$ to each party respectively.

It is clear that to recover original secret, S, *all* parties must cooperate by revealing their share to the other participants. Even in the event that a malicious party obtains $n - 1$ individual shares, they still gain no advantage towards recovering S in comparison to a party in possession of any other number of shares less than n. Continuing the above example, say a malicious party knew both s_1 and s_3, but not s_2. Recovering S with certainty that it is the original secret is impossible, as any guess made for s_3 has equal likelihood of being correct, as it was generated as a random value from $\mathbb{F}_{47}$. To recover S, we can simply perform a sum of all shares, $S = 12 + 25 + 4 = 41 \pmod{47}$.

2.2.3.2 Shamir Secret Sharing

In additive sharing schemes, such as the one described in Section 2.2.3.1, the restriction that *all* parties must cooperate to recover S can have severe limitation in certain contexts. For example, in the event that just one party is unable (or even just refuses) to participate, S will not be recoverable. A more flexible sharing scheme was introduced by Shamir in [16], known commonly as Shamir's Secret Sharing Scheme. Assuming that we again wish to share a secret S between n

parties, this scheme is more powerful in the sense that it can be adapted to allow for $t < n$ participants to cooperate to recover S. More specifically, Shamir defines this as a (t, n)-threshold sharing scheme, that is described by the following properties [16]:

- Any party in possession of t shares can recover S.
- However, any party in possession of $m < t$ shares is unable to recover S. Importantly, they are also unable to recover the remaining $t - m$ shares necessary to do so in their own.

To generate shares of a secret, S, Shamir's Secret Sharing Scheme requires the creation of a random polynomial, which is defined by Cartesian coordinates that represent shares of S. Recovering S requires the polynomial to be recovered via interpolation. Polynomial interpolation relies on the well-known theorem that, given $n + 1$ Cartesian points (with unique x-coordinates), there exists a unique polynomial of degree at most n that contains all $n+1$ points [18]. In other words, a polynomial of degree n requires at least $n+1$ known coordinates in order for it to be described sufficiently.

To utilise Shamir's Secret Sharing (t, n)-threshold Scheme, we must first pick a finite field, such as $\mathbb{F}_p$ for prime p, that our operations will be performed over. Then to share a secret, S, we perform the following as defined by Shamir [16]:

1. Generate a set of $t - 1$ random values, $\{a_1, \dots, a_{t-1}\}$, such that $a_i \in \mathbb{F}_p$. These will be coefficients of our polynomial. Furthermore, set $a_0 = S$.
2. Let $f(x) = \sum_{i=0}^{t-1} a_i x^i$.
3. Next, construct a set of n shares, $\{s_1, \dots, s_n\}$, such that $s_i = (i, f(i))$, for every $i \in \{1, \dots, n\}$.
4. Distribute each share s_i to the i-th party respectively. The polynomial $f(x)$ and its corresponding coefficients should be discarded.

As $f(x)$ is a degree $t - 1$ polynomial, it can be recovered with at least t points from $\{s_1, \dots, s_n\}$ via interpolation, using an algorithm of choice. Given f, the secret S can then be recovered by setting $S = f(0)$. A popular method to recover f is using Lagrange interpolation, which was first devised by Edward Waring [19]. The recovery of S using Lagrange interpolation is possible as follows, given any subset of at least t shares from $\{s_1, s_2, \dots, s_n\}$. Assuming that T is a set of the indices of the shares available to be used for recovery, then:

1. The function, f, can be reconstructed by setting $f(x) = \sum_{i \in T} y_i \lambda_i(x)$, where y_i is the y-coordinate of $s_i = (i, f(i))$ and $\lambda_i(x)$ is the *Lagrange coefficient* defined in our case as follows:

$$\lambda_i(x) = \prod_{\substack{j \in T \\ j \neq i}} \frac{x - j}{i - j}$$

2. The secret, S, can then be recovered by setting $S = f(0)$.

Any party with fewer than t shares cannot recover $f(x)$ and therefore S. This is because fewer than t points do not represent a $t-1$ degree polynomial uniquely. As Shamir notes in [16], this therefore means there are some $l > 1$ polynomials that are described by these points, all of which have equal probability of defining $f(x)$, as $\{a_1, \ldots, a_{n-1}\}$ is generated randomly.

To further illustrate this, consider the following example. Say we again wish to share the secret $S = 41$ between three parties, where any two can cooperate to recover S, independently from the other party. We choose our finite field to be $\mathbb{F}_{47}$ and perform the following to construct our shares using the Shamir Secret Sharing Scheme with a $(t = 2, n = 3)$-threshold:

1. Generate a set containing random values in $\mathbb{F}_{47}$ to represent our coefficients. Here we generate $\{33\}$, a set containing only a single value as our scheme has a threshold of two. Set $a_0 = 41$.
2. Let $f(x) = \sum_{i=0}^{t-1} a_i x^i = 41 + 33x$.
3. Pick $x_1 = 1$, $x_2 = 2$ and $x_3 = 3$. Therefore $s_1 = (1, f(1)) = (1, 74) \equiv (1, 27) \pmod{47}$. Similarly, $s_2 \equiv (2, 13) \pmod{47}$ and $s_3 \equiv (3, 46) \pmod{47}$.
4. Distribute shares s_1, s_2 and s_3 to each party respectively.

We can illustrate the recovery of S using only two shares and Lagrange interpolation, as defined earlier. Assuming a participant has obtained the shares s_2 and s_3, which means $T = \{2, 3\}$, therefore:

$$f(x) = 13 \cdot \frac{x-3}{2-3} + 46 \cdot \frac{x-2}{3-2}.$$

Therefore, we can recover S by setting $x = 0$:

$$S = f(0) = 13 \cdot \frac{0-3}{2-3} + 46 \cdot \frac{0-2}{3-2} = -53 \equiv 41 \pmod{47}.$$

Unlike garbled circuits, which were introduced by Yao for the explicit purpose of facilitating secure multiparty computation, it not immediately clear how secret sharing can be used to achieve the same goal. As such, the following example will demonstrate how Shamir Secret Sharing[1] can be used by three parties, P_1, P_2 and P_3 to jointly compute the function $f(x, y, z) = x + y + z$, whilst maintaining the privacy of their individual inputs x, y and z respectively. This concrete example is based on the general algorithm outlined by Smart in [20].

Assume that P_1, P_2 and P_3 wish to contribute $x = 41$, $y = 3$, $z = 18$ to the function $F(x, y, z)$ respectively. This can be jointly computed in a manner that maintains the privacy of each participants inputs as follows:

1. Each participant P_1, P_2 and P_3 generates shares of their private input using the general Shamir Secret Sharing algorithm described above (using their own defined function). As a result, in our example, each party generates the following shares (as before, we work in $\mathbb{F}_{47}$):

   P_1: (1,27), (2,13), (3,46) – generated with $f_1(x) = 41 + 33x$
   P_2: (1,14), (2,25), (3,36) – generated with $f_2(x) = 3 + 11x$
   P_3: (1,43), (2,21), (3,46) – generated with $f_3(x) = 18 + 25x$

2. Each participant, P_i, securely sends the share $(j, f_i(j))$ to participant P_j for $j \neq i \in \{1, 2, 3\}$ – that is, each participant sends a share to all other participants and keeps one share for themselves. As a result, each participant is in possession of the following shares:

   P_1: (1,27), (1,14), (1,43)
   P_2: (2,13), (2,25), (2,21)
   P_3: (3,46), (3,36), (3,46)

3. Each participant can then compute the sum of the y-coordinates of each share to obtain a Shamir share of the result of $F(41, 3, 18)$ as follows:

   P_1: $s_1 = 27 + 14 + 43 \equiv 37 \pmod{47}$
   P_2: $s_2 = 13 + 25 + 68 \equiv 12 \pmod{47}$
   P_3: $s_3 = 46 + 36 + 46 \equiv 34 \pmod{47}$

4. Now, because we have defined a (3,2)–threshold Shamir scheme, any two participants are required to interact to recover the output of $F(41, 3, 18)$. Assuming P_1 and P_2 send their share to each other, then they can each use Lagrange interpolation to recover the output of the joint computation without ever disclosing their input to the other participants:

$$12 \cdot \frac{0-3}{2-3} + 34 \cdot \frac{0-2}{3-2} \equiv 15 \pmod{47}$$

As required, this is exactly the same result as calculating $F(x, y, z) = 41 + 3 + 18 \equiv 15 \pmod{47}$, without any participant having to reveal their individual input.

[1]Note that both additive secret sharing and verifiable secret sharing can also easily be used to achieve secure multiparty computation in a similar manner.

Notice in this example, we performed secure multiparty computation involving three parties – this is not possible to achieve using garbled circuits as described in Section 2.2.2. Secret sharing schemes can also support secure function evaluation involving multiplication operations (e.g. a function such as $f(x, y, z) = xyz$), however this is a slightly more involved process than that which is demonstrated above. For a thorough description of this, see [20].

2.2.3.3 Verifiable Secret Sharing

Both Additive Secret Sharing and Shamir Secret Sharing rely heavily on two assumptions to ensure correctness. First, both schemes require a trusted dealer to distribute the shares of S. That is, all participants must trust that the dealer has distributed shares in a valid manner, such that each share can indeed be used to reconstruct S and is therefore not an arbitrary value that bears no mathematical relationship to S. Furthermore, all participants must trust that all other participants have submitted valid shares during the reconstruction of S, and again, not an arbitrary value that would cause the incorrect secret to be reconstructed.

To address these weaknesses, Chor et al. proposed what is known as Verifiable Secret Sharing (VSS) in [21]. VSS allows a secret S to be shared between n parties, such that the validity of each share received by a participant can verified, whether this be the share assigned to them by the dealer, or the share of another participant received during the reconstruction of S. A commonly known VSS scheme is defined by Feldman [22], that is an adaption of the Shamir Secret Sharing Scheme.

To achieve this, Feldman's scheme relies on the difficulty to compute the discrete logarithm (DLP) within certain groups. That is, given a finite cyclic group, G, with a generator g, and an element $h \in G$, there is no efficient algorithm known that can find an integer $0 \leq x < |G|$ such that $g^x = h$. Here, $|G|$ denotes the *order* of G, which is defined as the number of elements in G. Furthermore, g is a generator of G if all elements of the group can be written as a power of g (that is, each element of G can be obtained by repeatedly applying the group operation to g). A popular group for this purpose (and one that is chosen by Feldman in [22]) is the finite cyclic group, $\mathbb{Z}_p^*$, which is the multiplicative group of integers modulo a large prime, p. Given that all participants have agreed upon a group, $\mathbb{Z}_p^*$ and a generator, g, for this group, the Feldman VSS scheme can be used to share a secret, S, as follows:

1. The dealer generates a set of $t - 1$ random values, $\{a_1, \ldots, a_{t-1}\}$, such that $a_i \in \mathbb{Z}_p^*$. These will be coefficients of our polynomial. Furthermore, set $a_0 = S$.
2. The dealer sets $f(x) = \sum_{i=0}^{t-1} a_i x^i$.
3. The dealer constructs a set of n shares, $\{s_1, \ldots, s_n\}$, such that $s_i = (i, f(i))$, for every $i \in \{1, \ldots, n\}$.
4. The dealer distributes each share s_i to the i-th party respectively.

5. For every $0 \leq j \leq t-1$, the dealer also computes $\phi_j = g^{a_j}$, sending the set $C = \{\phi_0, \phi_1, \ldots, \phi_{t-1}\}$ to all participants. The polynomial $f(x)$ and its corresponding coefficients should then be discarded.
6. Upon receipt of s_i, each participant, P_i, can verify their share of S by computing $v = g^{s_i}$ and ensuring that the following equality holds:

$$v = \prod_{j=0}^{t-1} {\phi_j}^{i^j}$$

Clearly, the first four steps of Feldman's VSS scheme are simply a Shamir Secret Sharing of S, however the addition of step five and enables the share verification process that takes place in step six. Each element of C is a form of attestation to each coefficient of the function, f, without actually revealing the true value of said coefficients. This is possible due to the difficulty of the discrete logarithm problem, as described earlier. The equality in step six then allows the participant, P_i, to verify their share because:

$$\begin{aligned} v = \prod_{j=0}^{t-1} {\phi_j}^{i^j} &= \phi_0^{i^0} \phi_1^{i^1} \phi_2^{i^2} \ldots \phi_{t-1}^{i^{t-1}} \\ &= g^{a_0} g^{a_1 i} g^{a_2 i^2} \ldots g^{a_{t-1} i^{t-1}} \\ &= g^{\sum_{j=0}^{t-1} a_j i^j} \\ &= g^{f(i)} \\ &= g^{s_i} \end{aligned}$$

If the above equality does not hold, then P_i can be sure that they have not received a valid share of $S = a_0$ (and similarly the converse). A similar verification process takes place during the reconstruction of the secret, S. In Feldman's VSS scheme, each P_i participating in the reconstruction of S can verify the share, s_k, received from P_k by ensuring that the following equality holds:

$$g^{s_k} = \prod_{j=0}^{t-1} {\phi_j}^{k^j}$$

Once this equality has been verified for all shares required to reconstruct S, the secret can be obtained as per the reconstruction process defined in Section 2.2.3.2.

3 Applications & Implementations

In this section, we will introduce two of the most well-known real-world applications of MPC, namely privacy preserving auctions and privacy preserving analytics. For each application, we provide a high-level description of how the solution was achieved using the mathematical primitives of MPC described in Section 2.2. We then briefly describe three applications that could (potentially) be implemented commercially in the near future.

3.1 Applications of the Present

3.1.1 Privacy Preserving Auctions

Arguably the most well-known commercial application of MPC was developed and deployed in 2008, as a collaborative effort between SIMAP (a team of researchers from the University of Aarhus), DKS (the alliance of Danish sugar beet farmers), and Danisco (a Danish sugar production company) to facilitate a privacy preserving auction of sugar beets in Denmark [23]. This application was also the first practical implementation of an MPC protocol designed with the explicit aim of solving a real-world problem. A sugar beet is a crop grown by farmers for the production of sugar, which is extracted from the roots of the plant and refined before being distributed and sold. The MPC protocol proposed by Bogetoft et al. (SIMAP) in [23] allowed the sugar beet farmers and Danisco to calculate the market clearing price for the sale of sugar beets via a double auction, all while preserving the privacy of the bids submitted by each farmer.

The protocol devised by Bogetoft et al. is relatively straightforward in construction, requiring only the use of a verifiable secret sharing scheme from [24], an asymmetric encryption algorithm and a secure comparison protocol of their own design. The implementation described in [23] involves a joint computation between three servers over a local area network, where each server was owned by DKS, Danisco and SIMAP respectively. Each server receives a three-way Shamir secret shared input (a buy or sell bid) from each farmer, from which the total supply and demand at each price is calculated as a sum of all sell and buy bids at each price respectively. Then, using a binary search between the list of buy and the list of sell bids, the secure comparison function is used to calculate the list index (representing the unit price) where the difference between demand and supply is closest to zero. This computation is equivalent to finding the market clearing price of the sugar beet. Following the protocol description in [23], Bogetoft et al. prove that the protocol is secure against semi-honest adversaries, a security model that all cooperating parties agreed would be sufficient for their purposes.

The protocol of Bogetoft et al. achieves the principal objective of secure multi-party computation because the three participants were able to jointly compute the market clearing price of the sugar beet crop without the knowledge of the individual bids received from each farmer, and without the need for a trusted third party. As noted in [23], this was beneficial in two ways – first, using MPC allowed farmers to participate in the auction, without needing to disclose their bids to Danisco (which can be classed as sensitive information, because revealing these could allow Danisco to infer a farmer's financial situation). Secondly, the auction could take place without having to hire a third party consultant, reducing the cost to perform the auction for both DKS and Danisco.

Following on from this implementation, Partisia (a software solutions firm based in Denmark [25]) was founded by SIMAP, who have gone on to integrate MPC within a range of other products, including a privacy preserving survey platform [26], and a privacy preserving alternative financial trading system [27].

3.1.2 Privacy Preserving Analytics

Another well-known commerical implementation of secure multiparty computation is the Sharemind Framework [28], devised by Dan Bogdanov and Jan Willemson from Cybernetica (an Estonia-based technology company [29]) and Sven Laur from the University of Tartu. The Sharemind Framework differs from the work by Bogetoft et al. described in Section 3.1.1, as it is not an MPC protocol designed to address a specific real-world problem. Instead, it is runtime environment and programming language (named SecreC [30]) that allows developers to create privacy-preserving data analysis applications, without the need for the developer to understand the details of the underlying protocols and mathematical primitives. As outlined in [28], the operation of the Sharemind Framework is underpinned by additive secret sharing over $\mathbb{Z}_{2^{32}}$ which was chosen for efficiency reasons, as many modern 32-bit computers already implement integer arithmetic modulo 2^{32}. The Sharemind framework initially supported addition and multiplication over two secret shared values, along with greater-than-or-equal comparison in the semi-honest security model, however it has been shown that extensions to the original framework would also allow secure computation using Yao's garbled circuit protocol [31].

The Sharemind framework was first utilised to solve a real-world problem in 2011, when Bogdanov, Talviste and Willemson from Cybernetica developed a privacy-preserving application to analyse financial information for the members of the Estonian Association of Information Technology and Telecommunications (known as ITL) [32]. This allowed each member of ITL to compare financial indicators such as annual labour costs, training costs and profit with all other ITL members, without the need to share this data with them directly. As a result, it meant that the ITL members could reap the benefits of the analysis, while maintaining the privacy of their sensitive information, and without the requirement of a trusted third party (who may have abused their access rights to this information during the analysis). Similar to the implementation by Bogetoft et al. described in Section 3.1.1, the secure computation in [32] was performed between three servers in possession of Cybernetica, Microlink, and Zone Media respectively, all three of whom are members of ITL. These severs were assigned a three-way additive secret share of data submitted from each ITL member via a web application over the internet. Once the shares were received from each participant, each server performed the required analysis, which was essentially comprised of a privacy-preserving bubble sort (the code for which can be found in Appendix A of [33]). As a result, each server produced an additive share of the output, which was recombined with the output from the other two servers to obtain the analysis result, which was then used by the ITL members to generate reports for business stakeholders. As stated in [32], this application of the Sharemind Framework was the first time secure multiparty computation was used solve a real-world problem over a wide-area network.

Following on from this implementation, Cybernetica has acted as a consultant on numerous privacy-preserving data analytics projects using the Sharemind Framework, including:

- The development of an application for the US Defense Advanced Research Projects Agency (DARPA), to help estimate the probability of satellite collisions, with only the knowledge of secret shared satellite location data [34].
- A feasibility study, financed by the European Regional Development Fund, aimed to demonstrate that the Sharemind Framework could be used to perform genome-wide association studies, without the need for patients to disclose their sensitive medical records to a third party [35].

3.2 Applications of the Future

With the arrival of many practical implementations of MPC over the last decade, such as those described in Section 3, work has continued within academia to study the viability of employing MPC as a solution to a range of unique and interesting problems, such as:

- Using MPC to create privacy-preserving disease symptom trackers, such as that which is described in [36]. The underlying secure computation in [36] utilises SPDZ [37], a general purpose MPC protocol and implementation similar to Sharemind.
- Using MPC to train machine learning models without the requirement of full access to potentially sensitive data, as researched in [38]. The underlying secure computation described in [38] is supported by additive secret sharing.
- Using MPC to guarantee the privacy of the users of online dating platforms, as studied in [39]. The application of MPC in [39] is facilitated by Yao's garbled circuits.

4 Threshold Signature Schemes

4.1 Overview

Digital signature schemes – a class of cryptographic primitive providing both data origin authentication and non-repudiation – are heavily utilised within modern world processes, including digital communication (such as S/MIME [40]), software distribution, and cryptocurrency transactions (such as Bitcoin [41]). Digital signature schemes are generally comprised of three stages: key generation, message signing, and signature verification. Formally, following Katz and Lindell's definition [42], a digital signature scheme is comprised of a triple, (G, S, V), of probabilistic polynomial-time algorithms:

1. A Key Generation Algorithm, G – generates the public (verification) key, k_v, and private (signing) key[1], k_s. Any entity that executes this phase of the signature scheme must define the required *security parameter*, κ, often represented in unary as 1^κ. The security parameter, κ, is used to define the overall security of the scheme (e.g., the success probability of a potential adversary is measured as a function of κ). This algorithm is denoted: $(k_v, k_s) \leftarrow G(1^\kappa)$.
2. A Message Signing Algorithm, S – generates a signature, σ, computed over a message, $m \in M$, using the private key, k_s. Here, M is the *message space* defined within the signature scheme. This algorithm is denoted: $\sigma \leftarrow S_{k_s}(m)$.
3. A Signature Verification Algorithm, V – as input, takes a message, m, a signature, σ and public key, k_v, and as a result, generates a single-digit binary value, b. This is denoted: $b := V_{k_v}(m, \sigma)$.

For a digital signature scheme to be correct, it is required that for all κ, (k_v, k_s) generated by $G(1^\kappa)$ and $m \in M$:

$$V_{k_v}(m, \sigma) = 1$$

where $\sigma \leftarrow S_{k_s}(m)$. A signature, σ, is said to be *valid* if $V_{k_v}(m, \sigma) = 1$, and otherwise *invalid* if $V_{k_v}(m, \sigma) = 0$. At a high-level, a digital signature scheme can be used by an entity belonging to one of two categories. One party, known as a *signer* will execute the key generation algorithm, G, to obtain (k_v, k_s) and will publicly announce k_v. The signer will then compute $\sigma \leftarrow S_{k_s}(m)$ for a chosen message, m and publicise σ and m to any other entity that may wish to verify

[1]In this report, we use the terms public key and verification key interchangeably, and similarly private key and signing key interchangeably.

the signature at a later date. Any number of entities in possession of σ, m and k_v are then able to execute the signature verification algorithm, V, to determine if $V_{k_v}(m, \sigma) = 1$, and the signature is valid, or $V_{k_v}(m, \sigma) = 0$, and the signature is invalid. If the signature is valid, this means that the verifier can be sure that m was produced by the signer, it was not modified in transmission (data origin authentication) and the signer cannot later deny that the signature was created by them (non-repudiation).

For our purposes, the most important characteristic to observe in relation to traditional signature schemes, such as DSA and RSA (standardised in FIPS 186-4 [43]), is the one-to-many relationship between the number of signers and verifiers that can interact with the scheme. That is, all signatures created using traditional signature schemes can only be generated by a single signing party, yet any number of parties are able to verify this signature, given they are in possession of the signer's public key. Therefore, a natural extension of traditional signature schemes is to allow multiple signatories to participate in the signing process to produce a valid signature, akin to signatures in the physical world. This can be achieved using threshold signatures[2], the first of which was devised by Desmedt and Frankel [44] in 1991. Threshold signature schemes leverage secure multiparty computation techniques (namely, secret sharing schemes) to allow the signing parties to define a quorum of participants at the key generation phase, that must be present during the signing phase to produce a valid signature. Any number of signatories less than this defined quorum (or threshold) will not be able to produce a valid signature. As described throughout the literature [5, 44], a (t, n)–threshold signature scheme (with $t \leq n$) is a signature scheme that requires n participants to contribute during key generation and at least t of these participants to contribute during the signing phase to generate a valid signature. This is commonly known as a t-out-of-n threshold signature scheme. Formally, following Boldyreva's definition [45], a (t, n)–threshold signature scheme consists of a triple of polynomial-time algorithms / protocols, (TG, TS, V), that can be seen as an extension of the traditional signature scheme (G, S, V):

1. A Threshold Distributed Key Generation (DKG) Protocol, TG – an interactive protocol executed between $n > 1$ participants, $P_1, \ldots, P_n$, such that each participant, P_i, receives the public key, k_v, and a threshold secret share, s_i, of the corresponding private key, k_s. The share s_i is known only to participant P_i, and the private key, k_s is never known by *any* participant in its entirety.
2. A Threshold Message Signing Protocol, TS – an interactive protocol executed between all or a subset of $\{P_1, \ldots, P_n\}$, each of whom are in possession of their private share of k_s and a message m that each party has agreed to jointly sign. The execution of this protocol consists of two distinct phases: *signature share generation* and *signature construction*. To generate a signa-

[2]Threshold signatures are not the only technique that can be used to achieve this property, as will be discussed in Section 4.3.4.

ture share, each participant uses their share, s_i, of private key k_s to sign m and produce a share of the signature of m, denoted σ_i. To construct the signature, σ, each participant, P_i must send their signature share, σ_i to all other participants in the defined subset. Each participant, now in possession of a threshold of signature shares, can combine these to generate the signature, σ.

3. Signature Verification Algorithm, V – as input, takes a message, m, a signature, σ and public key, k_v, and as a result, generates a single-digit binary value, b. This is denoted: $b := V_{k_v}(m, \sigma)$. This algorithm is identical to the verification algorithm of a traditional signature scheme.

Throughout the rest of this report, we will refer to k_v as the *group verification key*, the corresponding private key, k_s, as the *group signing key*, and σ as the *group signature*. The crucial comparison we can make between a traditional signature scheme and a threshold signature scheme is that following a complete execution of TS, a group signature is derived that is indistinguishable from a traditional signature and can be verified using the group public key. As such, the verification algorithm in the threshold signature scheme does not require any modifications to be compatible with the verification algorithm of the underlying traditional signature scheme.

While the extension of traditional signature schemes to support the quorum property offered by threshold schemes is beneficial in most applications that employ digital signatures, as the utility and adoption of Bitcoin continues to increase, the necessity for the design and implementation of robust threshold signatures within Bitcoin is becoming more apparent. Indeed, the mismanagement of bitcoin custody (i.e., the loss of a private key) could lead to a loss of bitcoin with substantial financial value. As designed by Nakamoto in [41], Bitcoin transactions occur between users of the Bitcoin network and are authorised using digital signatures. Any user on the network can generate a public / private key pair – the private key authorises a transaction from the user by producing a signature over the transaction details (such as recipient address and amount), and the associated public key is used by nodes on the Bitcoin network to verify that the signature was indeed produced by the owner of the associated private key, and therefore whether they are authorised to spend the bitcoin specified in the transaction.

Two inherent weakness arise if Bitcoin transactions can only be approved by a single-party, the first relating to key management. A survey conducted in 2016 by Krombholz et al. [46] found that out of 990 participants, 22.5% admitted to having lost bitcoin, or a private key associated with bitcoin, at least once in the past. The reasons cited by Krombholz et al. include user error (such as hard drive formatting or a misplaced private key), security breaches, hardware failure, and software failure. Threshold signature schemes have the capacity to address this issue. A threshold signing key is produced in a distributed manner, such that each party contributes to the generation process, but no single party is ever in possession of the signing key in its entirety – and is never reconstructed – even during the signing process. It is in this sense that threshold signature schemes can be seen as an application of secure multiparty computation, where the key generation and signing phases are functions that multiple parties wish to jointly compute, without a trusted third-party nor the requirement for any parties' input (i.e., their contribution to the group's signing key) to be revealed to any other participant. Then, because threshold signature schemes are initialised with a threshold, (t, n), up to $n - t$ private keys can be lost or stolen, without a loss of funds. Illustrating this with a concrete example, assume that P_1, P_2 and P_3 each contribute to the key generation process, receiving their share of the group private key s_1, s_2, and s_3 respectively. Furthermore, assume that the participants have agreed to a threshold of two, meaning that they have agreed to participate in a $(2, 3)$-threshold signature scheme. It then follows that if only one party, P_i, loses their share, s_i for $i \in \{1, 2, 3\}$, the other parties P_j for $i \neq j \in \{1, 2, 3\}$ will still be able to produce a valid signature and therefore access their funds.

While similar to the first weakness described above, the second weakness in single-party signed transactions relates to shared custody and approval. Clearly, with transactions signed by a single party, only one party is required to sign and therefore approve a transaction. This, however, does not scale well in reality, where funds could be shared by a number of parties and should only be transferred when a defined quorum form an agreement. As an example, assume three business partners decide to jointly own bitcoin. If the private key is generated in the traditional single-party manner, all three parties would need to be trusted with the authority to perform transactions on their own. This trust relationship between parties can be exploited, if one of the business partners decides to perform a transaction against the will of the others, or is coerced into doing so by a malicious third-party. To address this issue, assume the three parties define a 3-out-of-3 threshold signature scheme to authorise their transactions. There is, therefore, no longer a requirement for trust between the signing parties, as all three parties must approve all transactions.

In the remaining sections of this chapter, we will explore the currently available commerical implementations of threshold signature schemes within Bitcoin *wallets* (software implementations that are utilised by users of the Bitcoin network to sign and therefore authorise transactions). We will then describe, in-detail, how a specific implementation of a contemporary threshold signature scheme (known as FROST [5]) achieves the desired properties described above. Included in this description, we will illustrate the utility of FROST by executing a proof-of-concept Python script that we have produced for this report. Finally, we will compare and contrast FROST with another signature scheme, namely MuSig2 [47], along with other mechanisms that can be used to achieve similar properties.

4.2 Commercial Implementations

Much of the innovation within the field of threshold signature schemes stems from researchers and developers working to implement commerical products. Sepior, a technology company based in Denmark claim to have been the first to implement a threshold signature cryptocurrency wallet and transaction solution in 2018 [48]. Since then, many more companies have implemented their own threshold signature wallets, including UnboundSecurity (formerly UnboundTech) [49], and ZenGo [50]. The products of both Sepior and Unbound Security are marketed for enterprise clients, and therefore not available to consumers. Both companies also have limited information available in relation to the exact technical specifications of their products. In contrast, ZenGo offer a consumer-grade threshold signature digital asset custody solution, known as the *ZenGo Wallet*, which is available on both the Apple App Store and Google Play Store. According to ZenGo's official website [50], the ZenGo wallet implements a modified version of Yehuda Lindell's threshold ECDSA signature scheme [51], where ECDSA is the elliptic curve implementation of the DSA algorithm, that has been standardised by NIST in FIPS 184-4 [43]. Lindell's protocol allows two participants to jointly generate an ECDSA key pair and signature, therefore translating to a 2-out-of-2 threshold signature scheme. This means that both participants must always interact to generate a valid signature (i.e., the threshold of the scheme is not user-defined). This protocol is suitable for ZenGo's use-case, which utilises Lindell's protocol in the following manner, as described in [50] and [52]:

1. Given that a user has downloaded and initialised the ZenGo Wallet mobile app, a Bitcoin public / private key pair is jointly generated by the user's mobile app and the ZenGo servers, such that each receive a share of the group private key, which itself is never in possession of either the mobile app or the ZenGo server.
2. Once both shares are generated, the mobile app will locally encrypt the user's share using the device's native key generation and encryption engine (e.g., the iOS Security Enclave) and send this encrypted share to ZenGo's servers for storage. The decryption key will then be stored by the user's cloud storage provider (e.g., iCloud). The decryption key is stored in the cloud to ensure that the user is still able to recover their share in the event that their mobile device is lost or damaged. Although ZenGo's servers are in possession both shares, ZenGo are unable recover the group private key and authorise transactions on their own, as they do not have access to the user's decryption key to reveal the user's share in plaintext.
3. To authorise a transaction from the ZenGo Wallet, the user first authenticates themselves to the app, which then fetches the decryption key from the cloud storage provider and the encrypted share from the ZenGo servers. The app can then recover the user's share and jointly sign and therefore authorise a transaction by executing the signing phase of Lindell's protocol [51] with the ZenGo servers.

ZenGo have also released an audited, open-source implementation of the threshold signature wallet described in [52], which is written in Rust and available at

[53]. This repository also includes three further implementations of alternative threshold ECDSA signature schemes [54, 55, 56], that have been developed since Lindell's protocol was released in 2017. The most recent of the three threshold signature schemes, proposed by Gennaro et al. [56], offers full t-out-of-n ECDSA signing capabilities and also provides *identifiable abort*, a mechanism that halts the protocol if misbehaving participants (i.e., those who do not participate in the protocol correctly) are detected, and publicises which participant caused the abort to other participants.

It is clear that a great deal of work has been done to not only develop efficient and secure threshold ECDSA schemes, but also integrate them into commercial products over the last few years. Threshold ECDSA has been the focus for many academics in the past, with it currently being the favoured signature scheme used to approve transactions within the Bitcoin network [57]. However, this is set to change in November 2021, when a soft fork (i.e., backwards compatible upgrade) known by developers as Taproot [58] will take place, affecting the format of transactions that will be deemed as valid on the Bitcoin network. Among other improvements, Taproot will allow Schnorr signatures to be used to sign transactions, alongside ECDSA [59]. As a result, there is now increased interest and research being undertaken to develop threshold Schnorr signature schemes, one of which is known as FROST [5].

4.3 FROST Signature Scheme

4.3.1 Traditional Elliptic Curve Schnorr

Before we delve into FROST, we must first describe how a traditional Schnorr signature scheme operates. The Schnorr signature scheme was first devised by Claus-Peter Schnorr in 1989 [60], however it did not receive the same acclaim and wide-spread adoption as DSA, as its use was restricted due to a patent that Schnorr filed in 1991 [61]. In his original paper [60], Schnorr designed the signature scheme to operate over the finite cyclic group, $\mathbb{Z}_p^*$, which is the multiplicative group of integers modulo a prime, p. The security of Schnorr, like many other signature schemes such as DSA, rely on the fact that there is no currently known method to efficiently solve the discrete logarithm problem (DLP), as we have already discussed in Section 2.2.3.3. The discrete logarithm problem can be applied to *any* finite cyclic group, and as a result, the Schnorr signature scheme can be modified to operate over an elliptic curve group, much like DSA has been extended to ECDSA. Commonly, an elliptic curve over a finite field, $\mathbb{F}_p$, is used and is denoted $E(\mathbb{F}_p)$, where p is a large prime. In this context, given a generator point, G, of the elliptic curve $E(\mathbb{F}_p)$, and a point Q on $E(\mathbb{F}_p)$, the DLP is the task of finding a integer, $0 \leq x < |E(\mathbb{F}_p)|$ such that $Q = x \cdot G$. As stated in [59], the Taproot soft fork to Bitcoin will implement the elliptic curve variant of the Schnorr signature scheme, operating over an elliptic curve known as *secp256k1*, which is recommended for cryptographic use by the Standards for Efficient Cryptography Group [62]. Various specifications currently exist to describe the elliptic curve variant of the Schnorr signature scheme, each with their own minor differences – for example, the ISO/IEC 14888-3 standard for elliptic curve Schnorr known as EC-FSDSA [63] and the specification outlined in BIP-340 [59] that will be implemented in Bitcoin, both differ with respect to the input of the hash function during the signing and verification phase of each scheme, however both are functionally equivalent. With a view to remain consistent with the FROST specification, our description of the Schnorr signature scheme will follow what is outlined in [5], however our description will extend this paper to operate over an elliptic curve, instead of in the group $\mathbb{Z}_p^*$.

Suppose that G is a generator point for $E(\mathbb{F}_p)$, where p is prime, and n is the order of G, which is also prime. Let $H : \{0,1\}^* \to \mathbb{F}_p$ be a hash function. Then, the Schnorr signature scheme is defined as a triple of probabilistic polynomial-time algorithms:

- A Key Generation Algorithm – given the security parameter (in unary) as input, 1^κ, this algorithm generates and returns the signing key, k_s, and verification point, Q, for the scheme as follows:
    1. Pick, at random, an integer $k_s \in [1, n-1]$ using the uniform distribution.
    2. Set $Q \leftarrow k_s \cdot G$.
    3. Return (k_s, Q).
- A Message Signing Algorithm – given, as input, a message $m \in \{0,1\}^*$ and signing key, k_s, this algorithm produces the signature, σ, over m as follows:
    1. Pick, at random, an integer $r \in [1, n-1]$ using the uniform distribution.
    2. Set $R \leftarrow r \cdot G$.
    3. Compute $c \leftarrow H(R \,\|\, Q \,\|\, m)$.
    4. Set $s \leftarrow c \times k_s + r \pmod{n}$.
    5. Return $\sigma = (R, s)$.
- A Signature Verification Algorithm – given a message $m \in \{0,1\}^*$, a verification point, Q, and signature, σ, as input, this algorithm returns a single-digit binary value, b, as follows:
    1. Parse σ as the components (R, s).
    2. Compute $c \leftarrow H(R \,\|\, Q \,\|\, m)$.
    3. Set $R' \leftarrow s \cdot G - c \cdot Q$
    4. Set $b \leftarrow 1$ if $R = R'$. Otherwise, set $b \leftarrow 0$.

5. Return b.

As the hash function, H, is defined to map an input of in the form $\{0,1\}^*$ to an element in $\mathbb{F}_p$, all inputs must be encoded into byte form. Both R and Q are points on an elliptic curve, and as such are comprised of an x-coordinate and a y-coordinate. Therefore, it is up to the implementer how these points are encoded. In EC-FSDSA [63], $R = R_x \parallel R_y$ where R_x and R_y are the x-coordinate and y-coordinate of R represented in bytes respectively. This is in contrast to BIP340 [59], that encodes R and Q using only their respective x-coordinates. Either way, it is only necessary that the signing and verification algorithm encode elliptic curve points in the same way to ensure that the scheme produces valid signatures. For our purposes, we will encode $R = R_x \parallel R_y$ and $Q = Q_x \parallel Q_y$ in both the signing and verification algorithms.

4.3.2 Specification & Operation

A number of threshold Schnorr signature schemes have been proposed since the beginning of the millennium, which [5] categories as either *robust* or *non-robust* schemes – a robust signature scheme successfully generates a valid signature in the presence of at most $n - t$ adversaries that contribute invalid shares, whereas non-robust schemes simply abort (i.e., fail) when an invalid share is included by an adversary. Early work in the field prioritised robust threshold Schnorr schemes, at the expense of a high number of communication rounds ([64] requires at least 4 rounds during the signature generation phase) or a strict requirement that a threshold of $t = n$ participants must be present to generate a valid signature (such as the scheme proposed by Gennaro et al. [65]). The Flexible Round-Optimized Schnorr Threshold (FROST) signature scheme, is a threshold Schnorr signature scheme, devised by Komlo et al. in 2020, and is non-robust. While this appears to be a downgrade from the previously proposed threshold Schnorr signature schemes, Komlo et al. state in [5] that in reality this allows for the scheme to be secure in the presence of a greater number of corrupt participants than in robust schemes – that is, FROST is secure for so long as the number of corrupt parties is not greater than or equal to the signing threshold, t, whereas robust schemes can only be secure if the number of corrupt participants does not exceed $n/2$ [66]. FROST improves upon both previously mentioned schemes, by requiring only one round of communication in the signing phase (in comparison to 4 round in [64]) and is able to produce a valid signature, as long as $t \leq n$ participants are present (unlike [65] that requires $t = n$ participants to produce a valid signature). Furthermore, the construction of FROST enables identifiable abort, unlike a scheme proposed by Abidin et al. [67] in 2019.

We will now provide a detailed description of the FROST signature scheme, first described in [5], with a step-by-step explanation as to how the scheme operates. We will describe the scheme over an elliptic curve, $E(\mathbb{F}_p)$ with order n, as opposed to group $\mathbb{Z}_p$ that is utilised in [5]. In line with our definition of threshold signature schemes from Section 4.1, FROST is comprised of two threshold algorithms, namely a threshold distributed key generation (TDKG) algorithm and a threshold signature algorithm. We will start by describing the TDKG algorithm.

4.3.2.1 Key Generation

At a high-level, the key generation algorithm of FROST is an extended form of Pedersen's distributed key generation [68], which in turn is based on Feldman's Verifiable Secret Sharing as described in Section 2.2.3.3, executed simultaneously by all n participants, such that no dealing party is required. As a result of this key generation algorithm, each party will obtain the group verification key, Q, and a Shamir share, s_i, of the corresponding signing key, k_s.

Assuming that the participants wish to define a t-out-of-n threshold scheme, each participant, P_i, constructs a polynomial f_i and a set of *commitments* to the coefficients of f_i, aligning with Feldman's VSS, as described in Section 4.3.1, albeit with minor modifications:

1. Each P_i must generate a set of $t-1$ random values, $\{a_{i0}, a_{i1}, \ldots, a_{i(t-1)}\}$, such that $a_{ij} \in \mathbb{F}_p$ is chosen using a uniform distribution.
2. Each P_i sets their function $f_i(x) = \sum_{j=0}^{t-1} a_{ij}x^j$.
3. For every $0 \leq j \leq t-1$, each P_i also computes $\Phi_{ij} = a_{ij} \cdot G$, sending the set $C_i = \{\Phi_{i0}, \Phi_{i1}, \ldots, \Phi_{i(t-1)}\}$ to all other participants.

As these steps of Feldman's VSS are executed by all n participants, the notation from Section 2.2.3.3 has been altered to reflect this. Here, a_{ij} denotes the j-th coefficient of P_i's polynomial, f_i, and Φ_{ij} denotes P_i's commitment to the coefficient, a_{ij}. In further contrast to Feldman's VSS, a_{i0} is generated uniformly at random from $\mathbb{F}_p$, not user defined as $a_{i0} = S$, for some predefined secret, S. Here, a_{i0} acts as P_i's contribution to the group signing key, and as such must be generated with suitable randomness.

Following this, each P_i must generate what is known as a *proof-of-knowledge* of their contribution, a_{i0}, to the group signing key and send this to all the other participants[3]. To do so, P_i must compute a standard Schnorr signature (as defined in Section 4.3.1) over their commitment $\Phi_{i0} = a_{i0} \cdot G$, as follows:

4. Each participant, P_i, picks, at random, an integer $k_i \in [1, n-1]$ using the uniform distribution.
5. Each P_i sets $R_i = k_i \cdot G$.

[3]This is required to prevent *rogue-key attacks*, which are well defined in [69]. Essentially, it ensures that participants cannot define their commitment as a function of the other participants commitments, which could allow signature forgeries.

6. Each P_i computes $c_i = H(i \parallel \Omega \parallel \Phi_{i0} \parallel R_i)$, where Ω has been included as a context string to prevent replay attacks using c_i.
7. Each P_i sets $\mu_i = c_i \times a_{i0} + k_i \pmod n$.
8. Each P_i sends $\sigma_i = (R_i, \mu_i)$ to all other participants.

Once each participant, P_i, has received a proof-of-knowledge from all other participants, they can verify them according to the standard Schnorr verification procedure as described in Section 4.3.1:

9. For every $1 \leq \ell \leq n$ such that $\ell \neq i$, each participant P_i computes $c_\ell = H(\ell \parallel \Omega \parallel \Phi_{\ell 0} \parallel R_\ell)$.
10. Each participant, P_i, then checks the following equality holds, for every $1 \leq \ell \leq n$ such that $\ell \neq i$:

$$R_\ell = \mu_\ell \cdot G - c_\ell \cdot \Phi_{\ell 0}$$

If, for any ℓ, the equality in step ten does not hold, the protocol will abort. The computations necessary to complete Feldman's VSS then resume, starting with Shamir share distribution and share verification:

11. For every $1 \leq \ell \leq n$, each P_i constructs a set of n shares, $\{s_{i1}, s_{i2}, \ldots, s_{in}\}$, of their contribution to the group signing key, a_{i0}, such that $s_{i\ell} = f_i(\ell)$.
12. Each P_i securely distributes the share, $s_{i\ell}$, to the ℓ-th participant respectively, keeping $s_{ii} = f_i(i)$ for themselves. The polynomial f_i and its corresponding coefficients should be discarded.
13. Every P_i now holds a set of Shamir shares, $\{s_{1i}, s_{2i}, \ldots, s_{ni}\}$, such that each $s_{\ell i}$ is the Shamir share of P_ℓ's contribution, $a_{\ell 0}$, to the group private key, k_s. To verify that each Shamir shares is valid, every P_i must compute the following, for $1 \leq \ell \leq n$ and $i \neq \ell$:

$$s_{\ell i} \cdot G = \sum_{j=0}^{t-1} i^j \cdot \Phi_{\ell j}$$

Again, if for any ℓ the equality does not hold, key generation aborts. The required signing keys and group verification key can then be computed as follows:

14. Each P_i can now compute their Shamir share, s_i, of the group signing key, k_s, by computing the following sum:

$$s_i = \sum_{j=1}^{n} s_{ji}$$

15. Each P_i can then also compute the group public key, Q, corresponding to the group signing key, k_s, by computing the following sum:

$$Q = \sum_{j=1}^{n} \Phi_{j0}$$

16. Finally, for reasons that will become clear in the signing protocol, each participant must also generate the share verification point for all other participants P_ℓ by computing the double sum:

$$Q_\ell = \sum_{j=1}^{n} \sum_{k=0}^{t-1} \ell^k \cdot \Phi_{jk} = \sum_{j=1}^{n} s_{j\ell} \cdot G = k_\ell \cdot G$$

In summary, this protocol has generated a group verification point, $Q = k_s \cdot G$, where $k_s = \sum_{i=1}^{n} a_{i0}$ is the group signing key. Each contribution by P_i to the signing key, a_{i0}, has been Shamir shared between n participants, such that P_i has been given the share $s_{\ell i} = f_\ell(i)$ of $a_{\ell 0}$, for each ℓ such that $1 \leq \ell \leq n$. The shares in possession of each participant are summed to obtain a single Shamir share of the group signing key, k_s. This follows in a similar fashion to the secure function evaluation example described in 2.2.3.2 – we know that $k_s = \sum_{i=1}^{n} a_{i0} = a_{10} + a_{20} + \ldots + a_{n0}$. Each contribution a_{i0} has been Shamir shared by P_i using the function:

$$f_i(x) = \sum_{j=0}^{t-1} a_{ij} x^j = a_{i0} + a_{i1}x + a_{i2}x^2 + \ldots + a_{i(t-1)}x^{t-1}.$$

For every ℓ such that $1 \leq \ell \leq n$, each P_i has been given the share $s_{\ell i}$ of $a_{\ell 0}$ from P_ℓ, where:

$$s_{\ell i} = f_\ell(i) = a_{\ell 0} + a_{\ell 1}i + a_{\ell 2}i^2 + \ldots + a_{\ell(t-1)}i^{t-1}.$$

As a result, each P_i is in the possession of a set of shares, $\{s_{1i}, s_{2i}, \ldots, s_{ni}\}$, and the following holds for each P_i:

$$\begin{aligned}
s_i &= s_{1i} + s_{2i} + \ldots + s_{ni} \\
&= f_1(i) + f_2(i) + \ldots + f_n(i) \\
&= \sum_{j=0}^{t-1} a_{1j} i^j + \sum_{j=0}^{t-1} a_{2j} i^j + \ldots + \sum_{j=0}^{t-1} a_{nj} i^j \\
&= \sum_{j=0}^{t-1} a_{j0} + \sum_{j=0}^{t-1} a_{j1} i + \ldots + \sum_{j=0}^{t-1} a_{j(t-1)} i^{t-1} \\
&= \alpha_0 + \alpha_1 i + \alpha_2 i^2 + \ldots + \alpha_{t-1} i^{t-1}.
\end{aligned}$$

Here, each α_ℓ is the sum of the coefficients of x^ℓ of the functions $f_\ell(x)$, for $1 \leq \ell \leq n$. Therefore, as by definition, $\alpha_0 = \sum_{j=0}^{t-1} a_{j0} = k_s$, it follows that s_i is a Shamir share of k_s that has been shared using the polynomial, $F(x)$, such that:

$$\begin{aligned}
F(x) &= \alpha_0 + \alpha_1 x + \alpha_2 x^2 + \ldots + \alpha_{t-1} x^{t-1} \\
&= k_s + \alpha_1 x + \alpha_2 x^2 + \ldots + \alpha_{t-1} x^{t-1}.
\end{aligned}$$

Therefore, s_i is simply the evaluation of $F(x)$ at the point $x = i$, which is exactly the construction of a Shamir share of k_s as defined in Section 2.2.3.2. We can then show that the generated verification point, Q, is indeed associated with k_s:

$$\begin{aligned}
Q &= \sum_{j=1}^{n} \Phi_{j0} \\
&= \Phi_{10} + \Phi_{20} + \ldots + \Phi_{n0} \\
&= a_{10} \cdot G + a_{20} \cdot G + \ldots + a_{n0} \cdot G \\
&= \left[\sum_{j=1}^{n} a_{j0} \right] \cdot G \\
&= k_s \cdot G
\end{aligned}$$

Following the above description, it is clear that this threshold distributed key generation protocol is indeed an application of MPC. Each participant, P_i has contributed a random value a_{i0} to the generation of k_s, which is the sum of $a_{\ell 0}$ for $1 \leq \ell \leq n$, computed by each P_i with *only* the knowledge of their contribution, a_{i0}. All other contributions, $a_{\ell 0}$ such that $1 \leq \ell \leq n$ with $i \neq j$, are kept private during the computation. What makes this an even more powerful application of MPC is the output, k_s, is also never known in full by any participant. Instead, only a Shamir share, s_i of the group signing key, k_s, is known by P_i. Furthermore, neither k_s nor any participant's contribution, a_{i0}, can be obtained during the computation of Q, as a result of the difficulty in solving the discrete logarithm problem.

4.3.2.2 Signing

At a high level, the threshold signing algorithm employed in FROST is comprised of two stages: signature share generation and signature construction. The signature generation phase is executed by each member of the defined subset of participants (the size of which must equal or exceed the threshold, t). As a result of this phase, each participant, P_i, will have generated a Shamir share, z_i, of the group signature, z, over the message m. Following this, a participant known as the *signature aggregator* (SA) will obtain a signature share, z_i, from each participant and combine them to produce a group signature σ over m. The role of the signature aggregator can be assigned to any participant, or even a third party, as the entity in this role does not gain any information that would compromise the security of the protocol (such as the group signature key). In the original paper [5], Komlo et al. define a *preprocess* phase that precedes the signature algorithm, which is included as a measure to increase the efficiency of the signature algorithm when it is performed more than once between the same set of participants. As we are only concerned with the operation of this algorithm to generate a single signature, we will not include the preprocess phase in our description, to avoid unnecessary complexities.

Assume that the key generation algorithm has been executed between n participants, and a subset, S, of n participants wish to jointly sign a message, m, where S contains the indices representing each participant. Again, the size of S must be at least t to produce a valid group signature. As a result, each participant P_i such that $i \in S$ is in possession of the group verification key, Q, and a Shamir share, s_i, of the group signing key, k_s. Furthermore, we define $H_1 : \{0,1\}^* \to \mathbb{F}_p$ and $H_2 : \{0,1\}^* \to \mathbb{F}_p$ to be hash functions. Then, FROST signature shares are generated as follows:

1. Each P_i such that $i \in S$ generates two integer nonces, $d_i, e_i \in [1, n-1]$, using the uniform distribution. In addition, each P_i generate commitments to these values, $D_i = d_i \cdot G$ and $E_i = e_i \cdot G$, sending (i, D_i, E_i) to all other participants, along with the message, m, they wish to sign.
2. Given each P_i receives (ℓ, D_ℓ, E_ℓ) from each participant such that $\ell \in S$ and $i \neq \ell$, each P_i must verify that every D_ℓ and E_ℓ lie on the curve $E(\mathbb{F}_p)$. Each P_i must also check that each m received is identical and matches the message they wish to sign. If either verification fails, the protocol aborts due to misbehaviour. Each participant then defines the set $B = \{(\ell, D_\ell, E_\ell)\}$ for all $\ell \in S$.
3. For every $\ell \in S$, each P_i computes the *binding values*[4], $\rho_\ell = H_1(\ell \| m \| B)$ and each participants contribution to the nonce commitment $R_\ell = D_\ell + (\rho_\ell \cdot E_\ell)$. By definition, this means that $R_\ell = r_\ell \cdot G$, where $r_\ell = (d_\ell + e_\ell \rho_\ell)$. The group nonce commitment can then be computed by each P_i as follows:

[4]These binding values are required to prevent Drijvers attack, as first described in [70]. An in-depth description of this attack can be found in [47].

$$R = \sum_{\ell \in S} R_\ell$$

The group nonce commitment, R, can equally be written $R = r \cdot G$, where $r = \sum_{\ell \in S} r_\ell$ is the group nonce, which is computed as by-product of the sum that is used to generate R above. The group nonce, r, cannot be computed directly by any P_i due to the difficultly of the discrete logarithm problem. With R now generated, we can proceed with signature share generation:

4. Each P_i computes the group challenge, $c = H_2(R \parallel Q \parallel m)$.
5. Then, to compute the group signature share, z_i, each participant, P_i, then computes the following:

$$z_i = c \times \lambda_i s_i + (d_i + e_i \rho_i) \pmod{n}$$

This last step is analogous to Step 4 of the traditional Schnorr signing algorithm in Section 4.3.1, where the term $(d_i + e_i \rho_i)$ can be seen as P_i's contribution to r, and $\lambda_i s_i$ can be seen as P_i's contribution to the signing key, k_s. Here, λ_i corresponds to P_i's Lagrange coefficient (defined in Section 2.2.3.2) evaluated at zero, which is used to convert P_i's Shamir share, s_i, of the group signing key, k_s, into an additive share of k_s. This Shamir to additive share conversion process is borrowed by the authors of FROST from Cramer et al. [71] and works similar to the recovery process for Shamir shared secrets in Section 2.2.3.2. It will become evident why this conversion process is necessary when signature construction is explained.

Given that each P_i is in now in possession of a Shamir share, z_i, of the group signature, z, the signature aggregator can now proceed with the signature construction phase of the protocol. We will assume that the signature aggregator has been chosen as one of the participants P_i of the signing protocol (therefore not a third-party). As such, the signature aggregator is already in possession of ρ_ℓ and R_ℓ for each $\ell \in S$, along with R and c. Then, the signature aggregator must perform the following:

6. Obtain z_ℓ from all $\ell \in S$, including their own signature share.
7. For each $\ell \in S$, verify that the following equality holds, where Q_ℓ is the share verification point generated in Step 16 of the key generation protocol:

$$z_\ell \cdot G = R_\ell + (c\lambda_\ell) \cdot Q_\ell$$

This verification step is performed to ensure that each participant, P_i, has generated a valid signature share and is therefore not attempting to subvert the protocol. If the signature share, z_ℓ has been generated correctly, this verification step holds because:

$$\begin{aligned} R_\ell + (c\lambda_\ell) \cdot Q_\ell &= D_\ell + \rho_\ell \cdot E_\ell + (c\lambda_\ell s_\ell) \cdot G \\ &= (d_\ell + \rho_\ell e_\ell) \cdot G + (c\lambda_\ell s_\ell) \cdot G \\ &= [cs_\ell \lambda_\ell + (d_\ell + e_\ell \rho_\ell)] \cdot G \\ &= z_\ell \cdot G \end{aligned}$$

8. Finally, the signature group signature, $\sigma = (R, z)$, can be published along with the message m by the signature aggregator, where z is computed as follows:

$$z = \sum_{\ell \in S} z_\ell$$

The simplicity of this final step is due to Shamir to Additive share conversion process that takes place in step five. By deconstructing this sum, we can see more clearly why this functions as intended:

$$\begin{aligned} z &= \sum_{\ell \in S} z_\ell \\ &= \sum_{\ell \in S} c\lambda_\ell s_\ell + (d_\ell + e_\ell \rho_\ell) \\ &= c \sum_{\ell \in S} \lambda_\ell s_\ell + \sum_{\ell \in S} (d_\ell + e_\ell \rho_\ell) \end{aligned}$$

We know from Section 4.3.2.1 that s_ℓ is simply P_ℓ's Shamir share of the group signing key, k_s, which is constructed from the function, $F(x)$. We also know that λ_ℓ is P_ℓ's Lagrange coefficient evaluated at zero. Therefore, from the Shamir share reconstruction process defined in Section 2.2.3.2, we know that the following holds:

$$\sum_{\ell \in S} \lambda_\ell s_\ell = F(0) = k_s$$

Furthermore, we know from step three of the FROST signing protocol the following is true:

$$\sum_{\ell \in S}(d_\ell + e_\ell \rho_\ell) = \sum_{\ell \in S} r_\ell = r$$

Here, r is the group nonce associated with the group nonce commitment $R = r \cdot G$, defined in the description following step three of the signing protocol. As a result, we can write the following:

$$\begin{aligned} z &= c\sum_{\ell \in S} \lambda_\ell s_\ell + \sum_{\ell \in S}(d_\ell + e_\ell \rho_\ell) \\ &= ck_s + r \end{aligned}$$

Therefore, given $z = ck_s + r$ and $R = r \cdot G$, it is clear that $\sigma = (R, z)$ is a valid signature on m with respect to the definition of a traditional Schnorr signature as defined in Section 4.3.1.

In summary, the signing protocol of FROST generates a traditional Schnorr signature $\sigma = (R, z)$ over m using the group signing key, k_s. To achieve this, a set of S participants must engage in the protocol, where $|S| \geq t$. As input, each participant P_i requires a Shamir share s_i of the group signing key, k_s, the group verification point, Q and the share verification point, Q_ℓ associated with P_ℓ for $\ell \neq i \in S$. To generate σ, the group signing key, k_s, is *never* reconstructed – indeed, the signing key is never reconstructed at any point during its lifecycle, even during key generation. This signing protocol is clearly an application of MPC. Given each signing participant P_i possesses a Shamir share, s_i of the group signature key, k_s, the protocol defined above allows the participants to jointly compute the output of the signing function (i.e., a Schnorr signature over m) while maintaining the privacy of each participants individual inputs (i.e., their Shamir share associated with k_s). As required by the definition of a threshold signature scheme from Section 4.1, there is no need to define a verification protocol to handle FROST signatures, as FROST's signature generation algorithm produces a traditional Schnorr signature as defined in Section 4.3.1. As such, the verification process follows exactly what is described in Section 4.3.1.

4.3.3 Proof-of-Concept Demonstration

To illustrate the feasibility and utility of the FROST signature scheme, we have produced a Python script that locally executes the key generation and signature generation protocols of FROST, with a view to demonstrate their correctness (i.e., the protocols produce the required output). Our implementation does not include any verification steps that are used to identify corrupt participants – namely, steps four to ten and step sixteen of the key generation algorithm (i.e., the proof-of-knowledge verification) and step two and seven of the signing algorithm. In a production environment, where these protocols are designed to executed on independent machines that interact over a network, verification steps are vital to ensure that all participants behave correctly. However, because our script is a proof-of-concept that is designed to perform all computations locally, as if to simulate the multiparty setting, including these verification steps would be unnecessary for our purposes.

Our implementation of FROST is comprised of three scripts:

- `EllipticCurve.py` – this script facilitates operations over an elliptic curve, including addition, subtraction, multiplication and equality comparison. We have defined our elliptic curve operations to take place over the secp256k1 curve. This script is listed in Appendix A.

- `FROSTLib.py` – this script contains all the functions necessary to perform key generation, along with signature generation and verification as we have outlined in Section 4.3.2. This script is listed in Appendix B.

- `FROST.py` – this script imports functions from `EllipticCurve.py` and `FROSTLib.py` so that FROST can be executed from a command line interface. This script is listed in Appendix C, which also defines which options can be specified during execution.

We will first demonstrate how our script can be used to generate a 3-out-of-5 threshold Schnorr signatures. Assuming that we have a terminal session open and are operating within a working directory that contains the three scripts listed above, FROST key generation can be performed as follows:

```
py FROST.py G `
-n 5 `
-t 3
```

Listing 4.1: FROST Key Generation using FROST.py

Here, the ` character allows us to define the terminal command over multiple lines. This command will output the following in the terminal:

```
----- Key Generation -----

Number of parties: 5
Singing threshold: 3

Group Verification (Public) Key:
(96260828981163801749224482766316882733140451526364848399529806302690869460221,
83965533227609309761895667960534248993548255020751096125821997624819009751889)

P_1's Shamir Share:
17826515639476480933797175472376571958193115222542062115252533256225420211144
P_2's Shamir Share:
15971723641373918405088824208877767447473703055712535250655233075858723271370
P_3's Shamir Share:
33768639448088533197708290832647764627853565780350367181047213343250109936474
P_4's Shamir Share:
36479636288425466684584279841080191143481434112733086591646925115944131758156
P_5's Shamir Share:
35683923086116338408073889735043837779641064480768183920714884708092604885848
```

Listing 4.2: Key Generation Output using FROST.py

As can be seen in Listing 4.2, each of the five participants receives a Shamir share, s_i, of the group signing key, k_s, which in turn is associated with the group verification key, $Q = k_s \cdot G$, which is a point that lies on the secp256k1 elliptic curve. In a production environment, the Shamir shares shown in Listing 4.2 would be generated between five separate entities (e.g., five servers), each generating a Shamir share of k_s that is only ever known to them and stored securely (such as on a hardware security module).

Now, assuming that three participants (say, P_2, P_3 and P_5, chosen arbitrarily) wish to jointly sign message (say, `"lorem ipsum"`), we can simulate the FROST signing protocol using the following command:

```
py FROST.py S `
-p 2 3 5 `
-s `
15971723641373918405088824208877767447473703055712535250655233075858723271370 `
33768639448088533197708290832647764627853565780350367181047213343250109936474 `
35683923086116338408073889735043837779641064480768183920714884708092604885848 `
-m "lorem ipsum" `
-q `
96260828981163801749224482766316882733140451526364848399529806302690869460221 `
83965533227609309761895667960534248993548255020751096125821997624819009751889
```

Listing 4.3: Signature Generation using FROST.py

As a result of executing this command, the following output is displayed in the terminal, which is a Schnorr signature of the form $\sigma = (R, z)$:

```
----- Signing -----

R:
(28509218713211509900654072927804769881578194825392330864789768854200657492853,
32069277231006561512927048381384137307384120915717722246959406290300608562277)

z: 1665511809140650853124441185559502373659226911195287706769967509309666535689
```

Listing 4.4: Signature Generation Output using FROST.py

The values of R and z will of course vary per execution, as a nonce value is freshly generated per signing operation. Finally, the signature over our example message can be verified using the following command:

```
py FROST.py V `
-r `
28509218713211509900654072927804769881578194825392330864789768854200657492853 `
32069277231006561512927048381384137307384120915717722246959406290300608562277 `
-z `
16655118091406508531244411855595023736592269111952877067699675093096665356589 `
-m "lorem ipsum" `
-q `
96260828981163801749224482766316882733140451526364848399529806302690869460221 `
83965533227609309761895667960534248993548255020751096125821997624819009751889
```

Listing 4.5: Signature Verification using FROST.py

Running the command in Listing 4.5 outputs the following the the terminal:

```
----- Verification-----

R_v:
(28509218713211509900654072927804769881578194825392330864789768854200657492853,
32069277231006561512927048381384137307384120915717722246959406290300608562277)

Signature Verified (R == R_v): True
```

Listing 4.6: Signature Verification Output using FROST.py

As can be seen in Listing 4.6, participants P_2, P_3 and P_5 have successfully produced a joint signature over the defined message that is valid according to the Schnorr verification algorithm described in Section 4.3.1. We can now demonstrate what occurs if the threshold of participants is not met when attempting to produce a joint signature using `FROST.py`. Assuming that each participant is in possession of the Shamir shares and group verification key present in Listing 4.2, we can observe the outcome of an execution of the signing protocol between P_2 and P_4. As before, a signature can be generated as follows:

```
py FROST.py S `
-p 2 4 `
-s `
159717236413739184050888242088777674474737030557125352506552330758587232713706 `
364796362884254666845842798410801911434814341127330865916469251159441317581563 `
-m "lorem ipsum" `
-q `
96260828981163801749224482766316882733140451526364848399529806302690869460221 `
83965533227609309761895667960534248993548255020751096125821997624819009751889
```

Listing 4.7: Invalid Signature Generation using FROST.py

This command returns the following signature in the terminal. As in our last example, the value of R and z here will vary per execution:

```
----- Signing -----

R:
(79711894534300787163933378548292328545558540962747370571669611131880115102677,
90698699280151876936264568357296771838873876527949883705873934222802404043645)

z: 41370641889203042384322102648861283734366258052854618888534681614446896180847
```

Listing 4.8: Invalid Signature Generation Output using FROST.py

We can then verify this signature in a similar manner to what was outlined in our previous example:

```
py FROST.py V `
-r `
79711894534300787163933378548292328545558540962747370571669611131880115102677 `
90698699280151876936264568357296771838873876527949883705873934222802404043645 `
-z `
41370641889203042384322102648861283734366258052854618888534681614446896180847 `
-m "lorem ipsum" `
-q `
96260828981163801749224482766316882733140451526364848399529806302690869460221 `
83965533227609309761895667960534248993548255020751096125821997624819009751889
```

Listing 4.9: Invalid Signature Verification using FROST.py

As a result of this command, the following is output in the terminal:

```
----- Verification-----

R_v:
(36698669569201395792937681337147276004439804300456925926697651498328826039677,
77628231457973356339637882711863783002051108424104675016208371779117408566432)

Signature Verified (R == R_v): False
```

Listing 4.10: Invalid Signature Verification Output using FROST.py

Referring back to Section 4.3.2.2, the signature in Listing 4.10 is invalid because an insufficient number of participants have contributed to the computation of z. In this case, only two of a required threshold of three participants have contributed to the signing process, and as a result the underlying group key, k_s, has not been reconstructed correctly to allow a valid Schnorr signature to be produced.

As we have clearly shown, the FROST signature scheme as described in Section 4.3.2 and originally by Komlo et al. in [5] does indeed satisfy two of the most fundamental properties of a threshold signature scheme, as defined in Section 4.1. That is, to produce a valid signature, a defined threshold of participants must contribute to the signing process, as seen in Listing 4.6. If the number of participants is below the defined threshold, the signature produced will not be valid, as seen in Listing 4.10. Furthermore, given the threshold of participants is met, FROST generates a valid signature according to the verification algorithm of the underlying traditional signature scheme, in this case being the Schnorr signature scheme.

The reader is welcome to use the scripts available in the Appendix at their discretion, either by confirming the computations outlined above, or to instantiate their own FROST signature scheme on their machine with custom parameters.

4.3.4 Alternative Techniques / Mechanisms

In the context of secure custody and transfer of digital assets (specifically Bitcoin in this report), there are a wide variety of mechanisms that can be used to achieve similar objectives to the FROST signature scheme. In this section, we will form a comparison between FROST and three alternative mechanisms, detailing how they can each be used to secure a Bitcoin wallet.

4.3.4.1 Shamir Secret Sharing

Although it may seem strange to compare FROST to a mathematical primitive that FROST itself depends upon to operate, to an extent Shamir Secret Sharing can be used independently to support the secure self-custody of bitcoin. Indeed, the popular hardware wallet manufacturer Trezor have implemented a variant of Shamir Secret Sharing into their Model T hardware wallet [72], which under Trezor's design documentation is known as SLIP-0039 [73]. This acts as a replacement for the BIP39 [74] design specification, which was created by the developers of Bitcoin to standardise a method for generating a backup of a Bitcoin wallet and the private key(s) associated with it using a human-readable mnemonic phrase comprised of 12 to 24 words. On the other hand, SLIP-0039 allows a wallet user to generate up to 16 distinct mnemonic phrases comprised of 20 to 33 words, each of which represent a shamir secret share of the Bitcoin private key[5]. Furthermore, the user can define the threshold of shares / mnemonic phrases required to recover the private key, which is performed using Lagrange interpolation, as described in

[5]In reality, the relationship between mnemonics phrases and private keys is slightly more complex than this (due to commonly used Hierarchical Deterministic Wallets [75]).

Section 2.2.3.2. In the event that any shares are lost or stolen, as long as a threshold of shares remain accessible to the legitimate owner of the wallet, the private key(s) and bitcoin associated with them will be recoverable.

Clearly, Shamir Secret Sharing is not a signature scheme and as such is not directly comparable to FROST. However, both mechanisms do interrelate, which is not surprising as the operation of FROST is itself underpinned by Shamir Secret Sharing. The crucial difference between a collection of wallets that (theoretically) implement FROST and a wallet that implements Shamir Secret Sharing as described in SLIP-0039, is that the latter is subject to a single point of failure. That is, if a Bitcoin private key is split using Shamir Secret Sharing alone, as implemented in the Trezor Model T, the private key must be recovered and present on a single device before any transactions can be signed. Further, the private key must also be present on a single device when the shares themselves are generated. As such, the device where private key generation or recovery takes place is a potential target for attack / theft by an adversary. Exacerbating this issue, SLIP-0039 actually only defines Shamir Secret Sharing as a backup technique – in other words, the Trezor Model T stores the private key in its entirety even after it has been split using Shamir Secret Sharing, therefore the device remains a single point of failure and target for attack, unless the device is factory reset after share generation.

In contrast, while the underlying operation of FROST relies on Shamir Secret Sharing, it is instead implemented in a distributed manner such that the private key is generated and Shamir shared simultaneously, and is therefore never present on a single device in its entirety. This is also the case during the signing protocol, whereby a joint signature is generated by a threshold of participants, such that a single entity is never in possession of group private key. Therefore, to gain unauthorised access to the group private key owned by a collection of wallets that implement FROST, an adversary would need to attack a threshold of wallets simultaneously to obtain a sufficient number of shares to recover the group private key and steal the bitcoin associated with it.

4.3.4.2 Native Bitcoin Multisignatures

The next alternative we will discuss is native Bitcoin multi-signatures (often referred to as *Multisig*). We use the term *native* multi-signature to refer to the multi-signature support that is offered directly by transactions that take place on the Bitcoin network. As outlined extensively in [57], all Bitcoin transaction contain embedded code, written in a language called *Bitcoin Script*, that is executed by nodes on the network to ensure they are valid (i.e, a transaction has been signed by the user that owns the associated bitcoin). This Script language features an *opcode* (i.e., a command) called *checkmultisig* that only allows the bitcoin associated with the transaction to be spent if a defined threshold of signatories include a valid signature of the transaction. This raises the question, is there any need for FROST (or indeed any other threshold signature scheme) to be implemented within Bitcoin wallets when transactions already seemingly support

t-of-n threshold signatures via Bitcoin Script? The answer depends on whether the two following benefits are important to the user:

- Reduced Transaction Fees – A multi-signature Bitcoin wallet with a threshold of t is essentially comprised of a n distinct wallets, each associated with their own public and private key. The bitcoin associated with a multi-signature wallet (i.e., unspent transactions) contain the public key associated with each of the n individual wallets. Then, to authorise a transaction from a multi-signature wallet, at least t of n participants must sign it separately, with their respective private keys. This means that size of the transaction (in bytes) scales in proportion to size of both n and t. As network usage fees depend on the number of bytes in a transaction, utilising native multi-signature can become costly if either t or n (or both) are large. As argued by Goldfeder et al. [76] a Bitcoin wallet implementing a threshold signature scheme (FROST, in our case) circumvents this issue, because the output of the signing protocol is a single signature, and as a result the size of a transaction would remain constant despite the size of n and t.
- Increased Privacy – In addition to reduced transaction fees, Goldfeder et al. also argue in [76] that because a threshold signature scheme produces a single signature as output, an external party observing the blockchain would be unable to distinguish between a transaction signed by a single party and one signed by multiple parties. This is in contrast to native Bitcoin multi-signatures because, as mentioned previously, each transaction must contain the public key of all n owners of the wallet, which could potentially reveal their identity.

4.3.4.3 MuSig2

MuSig is a multi-signature variant of the Schnorr signature scheme that was devised by Maxwell et al. [77] with the explicit aim of addressing the issues present with native Bitcoin multi-signatures as mentioned in Section 4.3.4.2. MuSig was first proposed in 2018, however the scheme was then succeeded with the release of MuSig2 by Nick et al. [47]. MuSig2 is largely identical to MuSig, however it improves upon the former by reducing the number of communication rounds required to produce a signature from three to two, and also addresses a flaw in the original scheme that meant it was susceptible to Drijvers attack, discovered by Drijver et al. in [70]. Both FROST and MuSig2 have the potential to reduce transaction fees and increase privacy on the Bitcoin blockchain should they ever be integrated into Bitcoin wallets in the future, however neither [47] nor [5] explain (at least, in detail) the fundamental differences between the schemes. As such, we will briefly describe the key generation and signing protocol of MuSig2, before comparing its underlying construction and functionality to FROST.

Similar with our description of FROST, we will extend the original specification of MuSig2 from [47] to operate over an elliptic curve, instead of within the group $\mathbb{Z}_p^*$. We define the elliptic curve $E(\mathbb{F}_p)$ with order n, where p is prime and G is a generator point for the curve. Furthermore, let $H_{\text{agg}}, H_{\text{non}}, H_{\text{sig}} : \{0,1\}^* \to \mathbb{F}_p$ be hash functions. The MuSig2 key generation protocol then operates as follows:

1. Each participant P_i picks, at random, an integer $k_i \in [1, n-1]$ using the uniform distribution.
2. Each P_i define their verification key as $Q_i \leftarrow k_i \cdot G$ and sends Q_i to all other participants.

Then, each participant computes the group verification key (a process which [47] refers to as *public key aggregation*):

3. Assume that n participants wish to generate a joint verification key and therefore each participant is in possession of the set, $L = \{Q_1, Q_2, \ldots, Q_n\}$, of individual verification keys from step two. Then, the group (or aggregate) verification key is computed by each P_i as follows:

$$Q = \sum_{i=1}^{n} a_i \cdot Q_i$$

Here, $a_i = H_{\text{agg}}(L \parallel Q_i)$ is known in [47] as the *key aggregation coefficient* and is included as a measure to prevent rogue-key attacks (again, which are defined in [69]). The signing protocol is then constructed as follows, beginning with shared nonce generation:

4. Each participant, P_i, picks v integers $r_{i1}, r_{i2}, \ldots, r_{iv} \in [1, n-1]$ at random using the uniform distribution. In [47], Nick et al. suggest a value of either $v = 2$ or $v = 4$.
5. Each P_i then computes $R_{ij} = r_{ij} \cdot G$ for each $j \in \{1, 2, \ldots, v\}$ and sends the set $\{R_{i1}, R_{i2}, \ldots, R_{iv}\}$ to all other signing participants.
6. Once each P_i has received a set of nonce values from all other signing participants, they can each compute the shared group nonces, $\{R_1, R_2, \ldots, R_v\}$, such that $R_j = \sum_{k=1}^{n} R_{kj}$ for all $j \in \{1, 2, \ldots, v\}$.

Using these nonces, each participant can now generate their individual component of the group signature of the message m:

6. Each participant P_i computes the group nonce as follows:

$$R = \sum_{j=1}^{v} b^{j-1} \cdot R_j$$

7. Using the group nonce, R, the group verification key, Q, and the message m, the group challenge can be defined by each participant:

$$c = H_{\text{sig}}(Q \parallel R \parallel m)$$

8. Each P_i computes their individual component of the group signature as follows, sending the output z_i to the signature aggregator (the same role as defined within FROST in Section 4.3.2.2):

$$z_i = ca_ik_i + \sum_{j=1}^{v} r_{ij}b^{j-1} \pmod n$$

Here, $a_i = H_{\text{agg}}(L \parallel Q_i)$ is the aggregation coefficient for P_i and k_i is the private key for P_i, as defined previously. In addition, $b = H_{\text{non}}(Q \parallel \{R_1, R_2, \ldots, R_v\} \parallel m)$. Finally, the group signature can be computed as follows:

9. The participant chosen as the signature aggregator can then compute the group signature $\sigma = (R, z)$, given they are in possession of individual signing component from all n signing participants, and where z is defined as follows:

$$z = \sum_{i=1}^{n} z_i$$

Now that we have given a brief description of MuSig2, we can now form a comparison between this scheme and FROST. We'll begin by describing the similarities between the two schemes:

- Both MuSig2 and FROST produce a single joint signature that can be verified using the standard Schnorr verification algorithm outlined in Section 4.3.1. In relation to signing Bitcoin transaction, this is clearly a desirable attribute as it means that nodes on the Bitcoin network only need to use one verification algorithm to validation all transactions on the blockchain, irrespective of whether the transaction was signed by one party or multiple parties.

- Both schemes are equally efficient with respect to the number of signing rounds – MuSig2 and FROST signing protocols are comprised of two communication rounds, which can be deconstructed to a pre-processing round and a signing round. Therefore, the signing protocols of both schemes only require a round of communication if the participants have previously signed a joint messaged and have opted to pre-compute nonces.
- Both MuSig and FROST are able to provide *concurrent security.* That is, when multiple signing sessions are opened simultaneously, both schemes are not susceptible to Drijvers attack. Furthermore, both schemes are also resistant to rouge-key attacks.

Finally, we will consider the differences between both signature schemes:

- The most significant difference both schemes is that FROST is a threshold t-out-of-n signature scheme, whereas MuSig2 is a n-out-of-n multi-signature scheme. In other words, a collection of n wallets that utilise the FROST protocol would be able to define a threshold $t \leq n$ such that only t or more participants are able to construct a valid signature. On the other hand, MuSig2 requires all n participants who contributed to the generation of the group verification key to participate in the signing process. FROST is able to offer this threshold property by utilising Distributed Key Generation to allow each participant to obtain a Shamir share, s_i, of the group private key without any participant holding the private key in its entirety. However, each participant using MuSig2 obtains an additive share, $a_i k_i$, of the group private key as a result of its key generation protocol. This additive share is not produced using distributed key generation – instead, it is simply a consequence of the group verification key generation process in step three of the MuSig2 specification. This becomes clear if we deconstruct this step as follows:

$$Q = \sum_{i=1}^{n} a_i \cdot Q_i = \sum_{i=1}^{n} a_i k_i \cdot G = (a_1 k_1 + \ldots + a_n k_n) \cdot G = k_s \cdot G$$

  Here, k_s is the group private key, therefore it is clear that $a_i k_i$ is the additive share of k_s owned by P_i. As explained at the beginning of Section 2.2.3.2, an additive secret sharing scheme is restrictive as it requires all parties with a share to participate in the recovery of the secret shared value. As a result, MuSig2 is a more restrictive signature scheme than FROST, as it requires all n parties to participate to produce a joint signature (due to the use of additive secret sharing), whereas FROST only requires that a threshold t participate (due to the adoption of Shamir secret sharing). Furthermore, FROST has the ability to function as a n-out-of-n signature scheme simply by defining the threshold $t = n$ during the key generation protocol – MuSig2 does not have this kind of flexibility.

- The process to generate individual signing and group (aggregate) verification keys is much simpler in MuSig2, in a sense that each participant can potentially generate them without real-time communication with the other participants. This is possible because the process to generate individual signing and verification keys is identical to the traditional Schnorr signature scheme, and the group verification key is simply a sum of the individual verification keys. Therefore, if a collection of parties wish to define a wallet that supports MuSig2 and each party already knowns the individual verification key of all other parties, the key generation phase can be performed without any communication between parties. In contrast, participants in FROST key generation must always engage in a round of communication with all other participants. This is because the individual signing keys are Shamir shares of the group singing key, which are generated using distributed key generation – a process that inherently requires communication.
- With respect to private key storage, MuSig2 suffers from a single point of failure. If a single private key associated with any of the participants wallets is lost or stolen, the bitcoin associated with the shared wallet will be irretrievable. This is because the group private key generation process utilises additive secret sharing. As explained previously, FROST does not suffer from this issues, as each participant is only in possession of a Shamir share of the group signing key, therefore as long as only a maximum of $n-t$ Shamir shares are lost or stolen, the bitcoin associated with a shared wallet implementing FROST will still be accessible.

4.3.4.4 Discussion

Now that we've compared FROST to three other mechanisms that offer similar benefits, can it be said that FROST is the perfect replacement to native Bitcoin multi-signatures, and further does it offer the best solution to the two inherent weaknesses of transactions signed by a single party? In other words, is FROST the best solution following the Bitcoin Taproot upgrade [59] to allow joint custody, in way that is also resilient to lost or stolen shares / private keys? While there isn't a clear-cut answer to these questions, as the term "best" is subjective and depends on the exact requirements of the user, it is clear that FROST offers an amalgamation of benefits from each three mechanisms, with very little downside. In summary, we will draw our together our discussion from the previous three subsection and make some final remarks.

Firstly, using Shamir Secret Sharing as described in Section 4.3.4.1 only offers benefits in the single user setting. While utilising Shamir Secret Sharing to split a mnemonic phrase clearly offers the user some reassurance that their bitcoin will be recoverable in the event that a share (or many) are lost or stolen, the fact that these shares must be recombined to sign a transaction means that the added security provided by sharing the private key is forfeited as soon as this is performed. Using Shamir Secret Sharing in this way also offers no solution to joint custody and transaction approval of bitcoin.

Next, native Bitcoin multi-signatures improve upon Shamir Secret Sharing in the sense that bitcoin associated with a multi-signature wallet can be held in joint custody by the participants that initially defined the wallet. Alongside this, transactions can be authorised without all private keys residing on a single device, eliminating the single point of failure present when utilising Shamir Sharing as described above. However, native multi-signatures are not private and can incur substantial transaction fees as the number of participants increases, which may or may not be an issue, depending on the user.

Finally, this leaves two options to consider – MuSig2 and FROST. From our discussion in Section 4.3.4.3, it is clear both schemes offer no obvious downsides that make one scheme vastly superior over the other. FROST is undoubtedly more flexible given that it allows for both t-out-of-n and n-out-of-n signature generation, whereas MuSig2 only allows n-out-of-n. Furthermore, FROST is much more resilient in the event of lost or stolen private keys / shares, due to its underlying use of Shamir secret sharing, as opposed to MuSig2 that utilises additive secret sharing. However, MuSig2 allows for non-interactive key generation, and as such it is more efficient and flexible in this regard. Therefore, if two wallets gave the option to use FROST or MuSig2, it is a clearly trade-off between efficiency or threshold support that a user must make when deciding which scheme to use.

5 Conclusion

In this project, we have first and foremost provided an introduction to the concept of secure multiparty computation, detailing how it was conceived and outlining its core objectives. Following this, we have provided a detailed description of three of the fundamental mathematical primitives that are commonly used to underpin the efficacy of MPC protocols. With a view to ensure that readers from a range of mathematical backgrounds are able to gain an understanding of these primitives, and how they can be used to achieve the core objectives of MPC, we have included worked examples and diagrams where appropriate. This builds upon the literature, which is often inaccessible to the average reader due an excessive use of mathematical proofs and a lack of concrete examples. To provide the reader with an appreciation for the practical utility of secure multiparty computation, we have also described two well-known real-world applications of MPC, illustrating how the mathematical primitives described in Section 2 have been used to facilitate privacy-preserving auctions [23] and privacy-preserving data analysis [28]. To complement this, we have also briefly provided examples of current research being undertaken into potential future applications of MPC. As a result, the first half of this report provides a comprehensive overview of secure multiparty computation, as required by our objectives.

Building on the foundation of knowledge gained from the first half of this report, we have provided a detailed discussion surrounding a particular application of MPC, namely threshold signatures. Similar to the structure of the previous sections, we first provided a overview of threshold signatures and how they differ from traditional signatures. We then described how threshold signatures schemes could be used to address two key issues in relation to digital asset self-custody, namely private key management and joint ownership of bitcoin. In addition to this, we also summarised the operation of the ZenGo wallet [50, 52], a commercially available Bitcoin wallet that utilises threshold ECDSA. With the aim of applying the knowledge attained in the first half of this report, we have then reviewed a state-of-the-art threshold Schnorr signature scheme, namely FROST [5], adding value to the subject area by performing the following tasks:

- We extended the specification from [5] to operate over an elliptic curve, to align with any potential future implementations of FROST within a Bitcoin wallet.
- We deconstructed the protocol specification from [5] to explain why certain steps are necessary and how FROST utilises MPC to achieve its goals.
- We produced a proof-of-concept implementation of FROST in Python – something that was not originally provided in [5].
- We compared FROST to three other techniques that could be (or are

currently being) used in Bitcoin wallet implementations, with a particular focus on the multi-signature signature scheme known as MuSig2.

As a result of this review, we have demonstrated the utility of FROST when applied to bitcoin custody and joint ownership, and have highlighted its potential benefits in comparison to other techniques, with a particular emphasis on MuSig2. Given the option to further extend our work on this project, we would spend the time improving our proof-of-concept Python implementation of FROST to include all verification steps and to potentially integrate a form of network communication functionality, so that FROST key generation and signing could take place on separate machines, as intended by the original specification in [5]. In the future, as the awareness of MPC increases, we expect to see continued growth in the number of applications implementing the techniques described in this report. However, there is still work to do if we are to ever see the mass adoption and implementation of threshold signature schemes, such as FROST, into Bitcoin wallets. This begins with the standardisation of threshold signature schemes, a process that is currently being undertaken by NIST in [78].

Bibliography

[1] A. C. Yao, "Protocols for secure computations," in *23rd annual symposium on foundations of computer science (sfcs 1982).* IEEE, 1982, pp. 160–164.

[2] Y. Lindell, "Secure multiparty computation for privacy preserving data mining," in *Encyclopedia of Data Warehousing and Mining.* IGI global, 2005, pp. 1005–1009.

[3] S. Micali, O. Goldreich, and A. Wigderson, "How to play any mental game," in *Proceedings of the Nineteenth ACM Symp. on Theory of Computing, STOC.* ACM, 1987, pp. 218–229.

[4] D. Bogdanov and O. Shlomovits, "NIST Multi-Party Threshold Schemes Workshop MPC Alliance Introduction," 2020. [Online]. Available: https://csrc.nist.gov/CSRC/media/Events/mpts2020/slides/mpts2020-2c5-brief-frank.pdf

[5] C. Komlo and I. Goldberg, "Frost: Flexible round-optimized schnorr threshold signatures." *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 852, 2020.

[6] D. Evans, V. Kolesnikov, and M. Rosulek, "A pragmatic introduction to secure multi-party computation," *Foundations and Trends in Privacy and Security*, vol. 2, no. 2-3, 2017.

[7] B. Hemenway, W. Welser, and D. Baiocchi, *Achieving Higher-Fidelity Conjunction Analyses Using Cryptography to Improve Information Sharing.* RAND Corporation, 2014.

[8] M. O. Rabin, "How to exchange secrets with oblivious transfer," Aiken Computation Lab, Harvard University, Tech. Rep., 1981.

[9] S. Even, O. Goldreich, and A. Lempel, "A randomized protocol for signing contracts," *Communications of the ACM*, vol. 28, no. 6, pp. 637–647, 1985.

[10] G. Brassard, C. Crépeau, and J.-M. Robert, "All-or-nothing disclosure of secrets," in *Conference on the Theory and Application of Cryptographic Techniques.* Springer, 1986, pp. 234–238.

[11] A. C. Yao, "How to generate and exchange secrets," in *27th Annual Symposium on Foundations of Computer Science (sfcs 1986).* IEEE, 1986, pp. 162–167.

[12] O. Goldreich, "Cryptography and cryptographic protocols," *Distributed Computing*, vol. 16, no. 2-3, pp. 177–199, 2003.

[13] D. Beaver, S. Micali, and P. Rogaway, "The round complexity of secure protocols," in *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, 1990, pp. 503–513.

[14] Y. Lindell and B. Pinkas, “A proof of security of yao’s protocol for two-party computation,” *Journal of cryptology*, vol. 22, no. 2, pp. 161–188, 2009.

[15] M. Naor, B. Pinkas, and R. Sumner, “Privacy preserving auctions and mechanism design,” in *Proceedings of the 1st ACM Conference on Electronic Commerce*, 1999, pp. 129–139.

[16] A. Shamir, “How to share a secret,” *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[17] G. R. Blakley, “Safeguarding cryptographic keys,” in *Managing Requirements Knowledge, International Workshop on.* IEEE Computer Society, 1979, pp. 313–313.

[18] J. F. Epperson, *An introduction to numerical methods and analysis.* John Wiley & Sons, 2021.

[19] E. Waring, “VII. Problems concerning interpolations,” *Philosophical transactions of the royal society of London*, no. 69, pp. 59–67, 1779.

[20] N. P. Smart, *Cryptography made simple.* Springer, 2016.

[21] B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch, “Verifiable secret sharing and achieving simultaneity in the presence of faults,” in *26th Annual Symposium on Foundations of Computer Science (sfcs 1985).* IEEE, 1985, pp. 383–395.

[22] P. Feldman, “A practical scheme for non-interactive verifiable secret sharing,” in *28th Annual Symposium on Foundations of Computer Science (sfcs 1987).* IEEE, 1987, pp. 427–438.

[23] P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter *et al.*, “Secure multiparty computation goes live,” in *International Conference on Financial Cryptography and Data Security.* Springer, 2009, pp. 325–343.

[24] I. Damgård and R. Thorbek, “Non-interactive proofs for integer multiplication,” in *Annual International Conference on the Theory and Applications of Cryptographic Techniques.* Springer, 2007, pp. 412–429.

[25] Partisia, “Homepage – Partisia,” https://partisia.com/, [Online; accessed 31-August-2021].

[26] Partisia, “Surveys – Partisia,” https://partisia.com/better-data-solutions/surveys/, [Online; accessed 31-August-2021].

[27] Partisia, “Secure Order Matching – Partisia,” https://partisia.com/better-market-solutions/order-matching/, [Online; accessed 31-August-2021].

[28] D. Bogdanov, S. Laur, and J. Willemson, “Sharemind: A framework for fast privacy-preserving computations,” in *European Symposium on Research in Computer Security.* Springer, 2008, pp. 192–206.

[29] Cybernetica, "Cybernetica," https://cyber.ee/, [Online; accessed 31-August-2021].

[30] Cybernetica, "sharemind-sdk/secrec: Sharemind SecreC Compiler and Analyzer," https://github.com/sharemind-sdk/secrec, [Online; accessed 31-August-2021].

[31] S. L. OlegŠelajev, "Yao garbled circuits in secret sharing-based secure multi-party computation," Cybernetica, Tech. Rep., 2011.

[32] D. Bogdanov, R. Talviste, and J. Willemson, "Deploying secure multi-party computation for financial data analysis," in *International Conference on Financial Cryptography and Data Security.* Springer, 2012, pp. 57–64.

[33] R. Talviste, "Deploying secure multiparty computation for joint data analysis—a case study," Master's thesis, University of Tartu, 2011.

[34] L. Kamm and J. Willemson, "Secure floating point arithmetic and private satellite collision analysis," *International Journal of Information Security*, vol. 14, no. 6, pp. 531–548, 2015.

[35] L. Kamm, D. Bogdanov, S. Laur, and J. Vilo, "A new way to protect privacy in large-scale genome-wide association studies," *Bioinformatics*, vol. 29, no. 7, pp. 886–893, 2013.

[36] A. Ranjan, S. Li, B. Chen, A. Chiu, K. Jagadeesh, and J. Liphardt, "Feveriq-a privacy-preserving covid-19 symptomtracker with 3.6 million reports," *medRxiv*, 2020.

[37] I. Damgård, V. Pastro, N. Smart, and S. Zakarias, "Multiparty computation from somewhat homomorphic encryption," in *Annual Cryptology Conference.* Springer, 2012, pp. 643–662.

[38] S. Yuan, M. Shen, I. Mironov, and A. C. A. Nascimento, "Practical, label private deep learning training based on secure multiparty computation and differential privacy," Cryptology ePrint Archive, Report 2021/835, 2021.

[39] J. Mikhail, E. Farag, E. Minasyan, and M. Romdhane, "Cryptographic Dating," https://courses.csail.mit.edu/6.857/2016/files/35.pdf, [Online; accessed 31-August-2021].

[40] J. Schaad, B. Ramsdell, and S. Turner, "Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 4.0 Message Specification," https://www.rfc-editor.org/rfc/rfc8551, 2019, [Online; accessed 31-August-2021].

[41] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized Business Review*, p. 21260, 2008.

[42] J. Katz and Y. Lindell, *Introduction to modern cryptography.* CRC press, 2020.

[43] National Institute of Standards and Technology, “Digital signature standard,” U.S. Department of Commerce, Washington, D.C., Tech. Rep. Federal Information Processing Standards Publications (FIPS PUBS) 186-4, 2013.

[44] Y. Desmedt and Y. Frankel, “Shared generation of authenticators and signatures,” in *Annual International Cryptology Conference.* Springer, 1991, pp. 457–469.

[45] A. Boldyreva, “Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme,” in *International Workshop on Public Key Cryptography.* Springer, 2003, pp. 31–46.

[46] K. Krombholz, A. Judmayer, M. Gusenbauer, and E. Weippl, “The other side of the coin: User experiences with bitcoin security and privacy,” in *International conference on financial cryptography and data security.* Springer, 2016, pp. 555–580.

[47] J. Nick, T. Ruffing, and Y. Seurin, “Musig2: Simple two-round schnorr multisignatures,” in *Annual International Cryptology Conference.* Springer, 2021, pp. 189–221.

[48] F. Wiener, “Threshold Signatures - Elevating Cryptocurreny Security,” https://static1.squarespace.com/static/586cf12903596e5605548ae1/t/5c6475ed7817f7bfb75d873d/1550087663616/Threshold+Signatures+-+Elevating+Crypto+Security.pdf, 2019, [Online; accessed 31-August-2021].

[49] Unbound Security, “CORE Crypto Asset and Blockchain Security — Unbound Security,” https://www.unboundsecurity.com/solutions/crypto-asset-security/, [Online; accessed 31-August-2021].

[50] ZenGo, “Security In-depth — ZenGo - Bitcoin & Cryptocurrency Wallet,” https://www.zengo.com/security-in-depth/, [Online; accessed 31-August-2021].

[51] Y. Lindell, “Fast secure two-party ecdsa signing,” in *Annual International Cryptology Conference.* Springer, 2017, pp. 613–644.

[52] ZenGo, “gotham-city/white-paper.pdf at master · ZenGo-X/gotham-city,” https://github.com/ZenGo-X/gotham-city/blob/master/white-paper/white-paper.pdf, [Online; accessed 31-August-2021].

[53] ZenGo, “GitHub - ZenGo-X/multi-party-ecdsa: Rust implementation of {t,n}-threshold ECDSA (elliptic curve digital signature algorithm).” https://github.com/ZenGo-X/multi-party-ecdsa, [Online; accessed 31-August-2021].

[54] R. Gennaro and S. Goldfeder, “Fast multiparty threshold ecdsa with fast trustless setup,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1179–1194.

[55] G. Castagnos, D. Catalano, F. Laguillaumie, F. Savasta, and I. Tucker, "Two-party ecdsa from hash proof systems and efficient instantiations," in *Annual International Cryptology Conference.* Springer, 2019, pp. 191–221.

[56] R. Gennaro and S. Goldfeder, "One round threshold ecdsa with identifiable abort." *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 540, 2020.

[57] A. M. Antonopoulos, *Mastering Bitcoin: unlocking digital cryptocurrencies.* O'Reilly Media, Inc., 2014.

[58] Bitcoin Core Contributors, "Bitcoin Core :: Bitcoin Core 0.21.1," https://bitcoincore.org/en/releases/0.21.1/, [Online; accessed 31-August-2021].

[59] P. Wuille, J. Nick, and T. Ruffing, "bips/bip-0340.mediawiki at master · bitcoin/bips," https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki, 2020, [Online; accessed 31-August-2021].

[60] C. P. Schnorr, "Efficient identification and signatures for smart cards," in *Conference on the Theory and Application of Cryptology.* Springer, 1989, pp. 239–252.

[61] C. P. Schnorr, "Method for identifying subscribers and for generating and verifying electronic signatures in a data exchange system," U.S. Patent 4,995,082, 1991.

[62] Standards for Efficient Cryptography Group (SECG), "Standards for efficient cryptography 2: Recommended elliptic curve domain parameters," Certicom Corp., Tech. Rep., 2010.

[63] International Organisation for Standardization, "ISO/IEC 14888-3: IT Security techniques — Digital signatures with appendix — Part 3: Discrete logarithm based mechanisms," 2018.

[64] D. R. Stinson and R. Strobl, "Provably secure distributed schnorr signatures and a (t, n) threshold scheme for implicit certificates," in *Australasian Conference on Information Security and Privacy.* Springer, 2001, pp. 417–434.

[65] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Secure applications of pedersen's distributed key generation protocol," in *Cryptographers' Track at the RSA Conference.* Springer, 2003, pp. 373–390.

[66] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Secure distributed key generation for discrete-log based cryptosystems," in *International Conference on the Theory and Applications of Cryptographic Techniques.* Springer, 1999, pp. 295–310.

[67] A. Abidin, A. Aly, and M. A. Mustafa, "Collaborative authentication using threshold cryptography," in *International Workshop on Emerging Technologies for Authorization and Authentication.* Springer, 2019, pp. 122–137.

[68] T. P. Pedersen, "A threshold cryptosystem without a trusted party," in *Workshop on the Theory and Application of of Cryptographic Techniques.* Springer, 1991, pp. 522–526.

[69] T. Ristenpart and S. Yilek, "The power of proofs-of-possession: Securing multiparty signatures against rogue-key attacks," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques.* Springer, 2007, pp. 228–245.

[70] M. Drijvers, K. Edalatnejad, B. Ford, E. Kiltz, J. Loss, G. Neven, and I. Stepanovs, "On the security of two-round multi-signatures," in *2019 IEEE Symposium on Security and Privacy (SP).* IEEE, 2019, pp. 1084–1101.

[71] R. Cramer, I. Damgård, and Y. Ishai, "Share conversion, pseudorandom secret-sharing and applications to secure computation," in *Theory of Cryptography Conference.* Springer, 2005, pp. 342–362.

[72] SatoshiLabs, "Shamir Backup - Our newest security standard — Trezor," https://trezor.io/shamir/, [Online; accessed 31-August-2021].

[73] P. Rusnak, A. Kozlik, O. Vejpustek, T. Susanka, M. Palatinus, and J. Hoenicke, "slips/slip-0039.md at master · satoshilabs/slips," https://github.com/satoshilabs/slips/blob/master/slip-0039.md, 2017, [Online; accessed 31-August-2021].

[74] M. Palatinus, P. Rusnak, A. Voisine, and S. Bowe, "bips/bip-0039.mediawiki at master · bitcoin/bips," https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki, 2013, [Online; accessed 31-August-2021].

[75] P. Wuille, "bips/bip-0032.mediawiki at master · bitcoin/bips," https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki, 2012, [Online; accessed 31-August-2021].

[76] R. Gennaro, S. Goldfeder, and A. Narayanan, "Threshold-optimal dsa/ecdsa signatures and an application to bitcoin wallet security," in *Applied Cryptography and Network Security*, 06 2016, pp. 156–174.

[77] G. Maxwell, A. Poelstra, Y. Seurin, and P. Wuille, "Simple schnorr multi-signatures with applications to bitcoin," *Designs, Codes and Cryptography*, vol. 87, no. 9, pp. 2139–2164, 2019.

[78] National Institute of Standards and Technology, "Multi-Party Threshold Cryptography — CSRC," https://csrc.nist.gov/projects/threshold-cryptography, 2021, [Online; accessed 31-August-2021].

Appendix

A EllipticCurve.py

```
class EllipticCurve():
def __init__(self,a,b,p,test=True):
        self.a = a % p
        self.b = b % p
        self.p = p
        self.test = test

        if (4*a**3 + 27*b**2) % p == 0:
                raise ValueError("Discriminant is zero modulo p.")

class Point():
        def __init__(self,curve,P):
                p = curve.p
                self.x = P[0] % p
                self.y = P[1] % p
                self.curve = curve

                if (curve.test):
                        self.__test__()

        def __test__(self):
                x = self.x
                y = self.y
                curve = self.curve

                if ((x**3 + curve.a*x + curve.b -y*y) % curve.p) != 0:
                        raise ValueError("This point is not on the specified curve.")

        def __repr__(self):
                return f"({self.x},{self.y})"

        def __eq__(self,Q):
                if (self.x == Q.x) and (self.y == Q.y):
                        return True

                return False

        def dbl(self):
                x = self.x
                y = self.y
                p = self.curve.p
                a = self.curve.a

                if(y == 0):
                        return Neutral(self.curve)

                den = pow(2*y,-1,p)
                m = (3*x*x + a)*den % p
                xd = (m*m - 2*x) % p
                yd = (-m*(xd - x) - y) % p

                return Point(self.curve,(xd,yd))

        def __add__(self,Q):
                if isinstance(Q,Neutral):
                        return self

                x1 = self.x
                y1 = self.y
                x2 = Q.x
                y2 = Q.y
                p = self.curve.p
                a = self.curve.a

                if x1 == x2:
                        if ((y1+y2) % p) == 0:
                                return Neutral(self.curve)

                        else:
                                den = pow(2*y1,-1,p)
                                m = (3*x1*x1 + a)*den % p

                else:
                        den = pow(x2 - x1,-1,p)
                        m = (y2 - y1)*den % p

                x3 = (m*m - x1 - x2) % p
                y3 = (-m*(x3 - x1) - y1)  % p

                return Point(self.curve,(x3,y3))

        def __neg__(self):
                return Point(self.curve,(self.x, self.curve.p-self.y))

        def __sub__(self,Q):
                return self + (-Q)

        def __rmul__(self,n):
                if n < 0:
                        return ((-n)*self).__neg__()

                if n == 0:
                        return Neutral(self.curve)

                Q = Neutral(self.curve)
                R = self

                # Double-and-add algorithm

                while n > 0:
                        if (n % 2 == 1):
                                Q = Q + R
                                n = n - 1
                        n = n // 2

                        if (n > 0):
                                R = R.dbl()

                return Q

        def get_x(self):
                return self.x

        def get_y(self):
                return self.y

class Neutral(Point):
        def __init__(self,curve):
                self.curve = curve

        def __eq__(self,Q):
                if isinstance(Q,Neutral):
                        return True

                return False

        def dbl(self):
                return self

        def __add__(self,Q):
                return Q

        def __sub__(self,Q):
                return Q.__neg__()

        def __neg__(self):
                return self

        def __rmul__(self,n):
                return self
```

B FROSTLib.py

```
from EllipticCurves import *
import secrets
from hashlib import sha256

rng = secrets.SystemRandom()

# The Secp256k1 prime

prime = int("FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F",
↪  16)

# The order of Secp256k1
```

```
order = int("FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141",
↪  16)

a = 0
b = 7

E = EllipticCurve(a,b,prime)

# Generate base point / generator of curve

base_x =
↪  int("79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798", 16)
base_y =
↪  int("483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8", 16)

G = Point(E,(base_x,base_y))

# ---------- Key Generation ----------

def generate_shamir_share(n,t):
    poly = [rng.randint(1,order) for party in range(t)]

    secret = poly[0]

    commit = secret*G

    shares = []

    for i in range(1,n+1):
        y_coord = sum([poly[j]*(i**j) for j in range(0,len(poly))]) % order

        shares.append(y_coord)

    return shares, commit

def assign_shares(shares_list):
    distributed_shares = [[] for i in range(len(shares_list))]

    for i in range(len(shares_list)):
        for j in range(len(shares_list)):
            distributed_shares[i].append(shares_list[j][i])

    distributed_shares = [sum(shares) for shares in distributed_shares]

    return distributed_shares

def generate_lambdas(party_indices):
    lambdas = []

    for party_index in party_indices:
        not_party_index = party_indices[:party_indices.index(party_index)] +
        ↪  party_indices[party_indices.index(party_index)+1:]

        numerator = 1
        denominator = 1
        for index in not_party_index:
            numerator = numerator * index
            denominator = denominator * (index - party_index)

        denominator = pow(denominator,-1, order)

        lambdas.append((numerator * denominator) % order)

    return lambdas

def generate_keys(n,t):
    shamir_shares = []
    commitments = []

    for party in range(n):
        share, commit = generate_shamir_share(n,t)
        shamir_shares.append(share)
        commitments.append(commit)

    private_keys = assign_shares(shamir_shares)

    public_key = Neutral(E)

    for commit in commitments:
        public_key += commit

    return public_key, private_keys

# ---------- Signing ----------

def H_1(l,m,B):
    l_bytes = int.to_bytes(l, 32, "big")
    m_bytes = m.encode("utf-8")
    B_bytes = bytes()

    for commitment_nonce in B:
        for value in commitment_nonce:
            B_bytes += int.to_bytes(value, 32, "big")

    hash = sha256(l_bytes+m_bytes+B_bytes).digest()

    return int.from_bytes(hash, "big") % order

def H_2(R,Y,m):
    R_bytes = int.to_bytes(R.get_x(), 32, "big")
    Y_bytes = int.to_bytes(Y.get_x(), 32, "big")
    m_bytes = m.encode("utf-8")

    hash = sha256(R_bytes+Y_bytes+m_bytes).digest()

    return int.from_bytes(hash, "big") % order

def generate_nonces(party_indexes):
    nonces = []
    for party_index in party_indexes:
        d = rng.randint(1,order)
        e = rng.randint(1,order)
        D = d*G
        E = e*G

        nonces.append([[d,D],[e,E]])

    return nonces

def generate_B(nonces, party_indexes):
    B = []

    for i in range(len(party_indexes)):

        D = nonces[i][0][1].get_x() # The x-coord is chosen.
        E = nonces[i][1][1].get_x()

        B.append([party_indexes[i], D, E])

    return B

def sign(m,participants,private_keys,group_public):
    nonces = generate_nonces(participants)

    B = generate_B(nonces, participants)

    rhos = []

    for party_index in participants:
        rhos.append(H_1(party_index,m,B) % order)

    R = Neutral(E)

    for j in range(len(participants)):
        D_i = nonces[j][0][1]
        E_i = nonces[j][1][1]

        R += (D_i + (rhos[j]*E_i))

    c = H_2(R,group_public,m) % order

    lambdas = generate_lambdas(participants)

    z_shares = []

    for j in range(len(participants)):
        d_i = nonces[j][0][0]
        e_i = nonces[j][1][0]
        rho_i = rhos[j]

        lambda_i = lambdas[j]
        key_i = private_keys[j]
        z_shares.append((d_i + e_i*rho_i) + lambda_i*key_i*c % order)

    return R, sum(z_shares) % order

def verify(R,z,m,group_public):
    c = H_2(R,group_public,m) % order

    R_v = (z*G)+((-c)*group_public)

    return R_v
```

C FROST.py

```
from FROSTLib import *
from EllipticCurves import *
import argparse

# The Secp256k1 prime

prime = int("FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F",
↪  16)

# The order of Secp256k1

order = int("FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141",
↪  16)

a = 0
b = 7

E = EllipticCurve(a,b,prime)

parser = argparse.ArgumentParser(description="Proof-of-concept implementation
↪  of FROST signature scheme.")
parser.add_argument("mode", type=str, help="Indicate the mode FROST should
↪  execute in -- i.e., key generation (G), signing (S), verification (V) or
↪  both (A).")
parser.add_argument("-n", type=int, help="Number of entities participating in
↪  key generation.")
parser.add_argument("-t", type=int, help="The number (threshold) of entities
↪  that must participate in the signing protocol to generate a valid
↪  signature.")

parser.add_argument("-s", type=int, nargs='+', help="The Shamir shares of each
↪  participant (in order) associated with the group signing key, specified as
↪  space seperated integers.")
parser.add_argument("-m", type=str, help="The message to be signed.")
parser.add_argument("-p", type=int, nargs='+', help="The indices representing
↪  the participants that wish to jointly sign the message, specified as a
↪  sequence of space seperated integers.")
parser.add_argument("-q", type=int, nargs='+', help="The group verification
↪  key, specified as space seperated integers, the first and second integer
↪  representing the x-coord and y-coord respectively (e.g. R_x R_y)")
```

```
parser.add_argument("-r", type=int, nargs='+', help="The group nonce, specified
↪  as space seperated integers, the first and second integer representing the
↪  x-coord and y-coord respectively (e.g. Q_x Q_y).")
parser.add_argument("-z", type=int, help="The group signature response, z.")

args = parser.parse_args()

# Assign user inputs to variables

mode = args.mode
parties = args.n
shares = args.s
threshold = args.t
message = args.m
participants = args.p
group_response = args.z
group_public = args.q
nonce = args.r

if args.q is not None:
    group_public = Point(E,(group_public[0],group_public[1]))

if args.r is not None:
    nonce = Point(E,(nonce[0],nonce[1]))

if mode == "A":
    if parties is None or threshold is None or participants is None or message
    ↪  is None:
        print("\nThis mode ("+mode+") of FROST requires the following
        ↪  parameters: n, t, p, m. Type script.py -h for more information.\n")
        exit()
elif mode == "G":
    if parties is None or threshold is None:
        print("\nThis mode ("+mode+") of FROST requires the following
        ↪  parameters: n, t. Type script.py -h for more information.\n")
        exit()
elif mode == "S":
    if shares is None or message is None or participants is None or
    ↪  group_public is None:
        print("\nThis mode ("+mode+") of FROST requires the following
        ↪  parameters: p, s, m, q. Type script.py -h for more information.\n")
        exit()
elif mode == "V":
    if nonce is None or group_response is None or message is None or
    ↪  group_public is None:
        print("\nThis mode ("+mode+") of FROST requires the following
        ↪  parameters: r, z, m, q. Type script.py -h for more information.\n")
        exit()
else:
    print("\nThis mode ("+mode+") is not valid - please specify key generation
    ↪  (G), signing (S), verification (V) or all (A) mode.\n")
    exit()

if mode == "G" or mode == "A":
    # ----- Key Generation -----

    group_public, shares = generate_keys(parties,threshold)

    # ----- Terminal Output -----

    print("\n")
    print("----- Key Generation -----")
    print("\n")
    print("Number of parties:", parties)
    print("Singing threshold:", threshold)
    print("\n")
    print("Group Verification (Public) Key:", group_public)
    print("\n")

    for party in range(parties):
        print("P_"+str(party+1)+"'s Shamir Share:", shares[party])

    print("\n")



    print("\n")

if mode == "S" or mode == "A":

    # ----- Signing -----

    signing_keys = []

    if mode == "A":
        for party_index in participants:
            signing_keys.append(shares[party_index-1])
    else:
        signing_keys = shares

    nonce, group_response =
    ↪  sign(message,participants,signing_keys,group_public)

    # ----- Terminal Output -----

    if mode == "S":
        print("\n")

    print("----- Signing -----")
    print("R:", nonce)
    print("z:", group_response)
    print("\n")

if mode == "V" or mode == "A":
    # ----- Verification -----

    nonce_v = verify(nonce,group_response,message,group_public)
```

```
# ----- Terminal Output -----

if mode == "V":
    print("\n")

print("----- Verification-----")
print("R_v:", nonce_v)
print("Signature Verified (R == R_v):", nonce == nonce_v)
print("\n")
```