Bitcoin PIPEs

Covenants and ZKPs on Bitcoin Without Soft Fork

Mikhail Komarov
nemo@allocin.it
[[alloc] init]

May 1, 2024

Abstract

Succinct zero-knowledge proofs verification on the Bitcoin L1 has long been considered unfeasible due to the limitations of the existing Bitcoin Script language. Specifically, the absence of covenants, such as CAT, CTV, $CSFS$ (and a small upper limit on the script size), has prevented the implementation of Merkle tree paths verification required for FRI/LPC-alike commitment schemes commitments verification along with arithmetization definitions computations requiring OP_MUL to be enabled. Despite various proposals (e.g. BIP-347) to re-enable or introduce new covenant opcodes, these changes have not been adopted, leaving ZKP verification on the Bitcoin L1 an unresolved challenge.

This paper proposes using Bitcoin PIPEs for verifying ZKPs on the Bitcoin L1 - an approach to enable a Polynomial Inner Product Encryption (PIPE)-based SNARK verification on Bitcoin (starting with Placeholder proof system [1]) with (i) emulating abscent covenants (e.g. CAT) through the usage of Bitcoin PIPEs framework and (ii) by introducing Bitcoin PIPE for Placeholder proof system verification itself (effectively introducing the Placeholder verification FH-MIPE-based covenant opcode with this). The method proposed involves generating unique keys and signatures that are conditionally valid based on the satisfaction of Placeholder proof conditions. This approach not only overcomes the current limitations of Bitcoin Script but also opens up new possibilities for implementing new kinds of applications on the Bitcoin L1 (via application-specific Bitcoin PIPEs covenants) alongside true Bitcoin zkRollups.

1 Introduction

Zero-Knowledge Proofs verification on the Bitcoin L1 has long been considered unfeasible due to the limitations of the existing Bitcoin Script language. Specifically, the absence of covenants, such as CAT, CTV, $CSFS$ (and a small upper limit on the script size), has prevented the implementation of Merkle tree paths verification required for FRI/LPC-alike commitment schemes commitments verification along with arithmetization definitions computations requiring OP_MUL to be enabled. Despite various proposals (e.g. BIP-347) to re-enable or introduce new covenant opcodes, these changes have not been adopted (yet?), leaving ZKP verification on the Bitcoin L1 an unresolved challenge.

The obvious solution to this problem is to upgrade Bitcoin's protocol, to introduce (or re-introduce) absent opcodes. This unfortunately leads to the necessity to achieve social consensus, which is a quite complicated process. This means that the next solution in line is to emulate covenants necessary for particular application.

This paper proposes an approach to (i) define application-specific covenants on Bitcoin by leveraging FH-MIPE predicates and (ii) enable a hash-based commitment scheme (LPC) proof system (Placeholder[1]) proofs verification on the Bitcoin L1 via (i) emulating abscent covenants (e.g. CAT) through the use of Function Hiding Multi-Input Predicate Encryption (FH-MIPE) and (ii) by introducing FH-MIPE predicate-defined Placeholder verification covenant (effectively introducing the Placeholder verification covenant opcode with this). The method proposed involves generating unique keys and signatures that

are conditionally valid based on the satisfaction of Placeholder proof conditions. This approach not only overcomes the current limitations of Bitcoin Script but also opens up new possibilities for implementing new kinds of applications on the Bitcoin L1 (via application-specific FH-MIPE covenants) alongside with true Bitcoin zkRollups.

2 Preliminaries

2.1 Covenants

Covenants are restrictions on Bitcoin transactions. They define rules about where and how Bitcoin can be spent, adding a layer of programmability to Bitcoin transactions. Covenants are not currently part of Bitcoin's native functionality and require the community to agree on and implement specific upgrades. Several notable attempts were made recently to introduce controversial covenants opcodes (without any luck for now unfortunately):

1. **CheckTemplateVerify (CTV)**: *CTV* is a covenant which only allows the exact next transaction to be executed. It allows a user to commit to a specific transaction by ensuring that only the hash of that transaction matches a predefined value. It was proposed back in 2020 and got assigned BIP-119.

2. **Concatenation (CAT)**: *CAT* operation is crucial for more advanced covenants. It concatenates two data items on the stack, enabling more complex scripts and conditions. For instance, *CAT* can be used to implement covenants that check multiple conditions on Bitcoin being spent, allowing for a broader range of transaction types and restrictions.

   CAT's significance lies in its ability to support complex Bitcoin Script operations that go beyond simple locking and unlocking scripts. This makes it possible to create more sophisticated covenants that can enforce a wide variety of spending conditions.

   In particular *CAT* can help construct Merkle trees for verifying commitments of hash-based commitment scheme-based proof systems proofs within Bitcoin script natively.

   There were also a BIP introduced to reflect *CAT*: BIP-347.

2.1.1 Covenants: *CAT*

The absence of concatenation covenant (expressed in an `OP_CAT` opcode) in Bitcoin has made certain operations cumbersome or impossible. `OP_CAT` is essential for efficient Merkle tree operations for verifying commitments. Without `OP_CAT`, simulating its functionality requires cumbersome workarounds that are often impractical. The re-enabling of `OP_CAT` would simplify these operations and make Bitcoin scripts more powerful and flexible.

The re-enabling of `OP_CAT` has been positively received by the Bitcoin community, with several proposals and discussions taking place to bring it back. The draft BIP-347 (`https://github.com/bitcoin/bips/blob/master/bip-0347.mediawiki`) for `OP_CAT` has undergone several iterations, and its implementation in Bitcoin Core is actively being discussed. Re-enabling `OP_CAT` would require a soft fork, which, if successful, would mark a significant enhancement in Bitcoin's scripting capabilities.

The progress towards re-enabling `OP_CAT` is promising, with discussions and reviews happening in bitcoin-dev mailing list. The potential activation of `OP_CAT` would enable more advanced scripts and applications on Bitcoin, paving the way for Turing-complete applications and improved functionality.

2.1.2 Bitcoin-friendly Proof Systems

To verify zero-knowledge proofs on Bitcoin, the proof system must be efficient and fit within Bitcoin's constraints. A Bitcoin-friendly proof system should minimize the weight units used in the script, stay within stack limits, and utilize existing opcodes like hash functions to reduce computational costs. Hash-based commitment scheme-enabled proof systems, such as Placeholder, are more likely to be compatible with Bitcoin due to their reliance on hash functions and a more ofter usage of smaller prime fields.

By leveraging recursive verification and optimizing for Bitcoin's limitations, it is possible to create proof systems that are both efficient and practical for on-chain verification. The combination of `OP_CAT` and efficient proof systems can enable powerful and flexible covenants, enhancing Bitcoin's programmability and privacy features.

Unfortunately, `OP_CAT` being enabled will take time. This means to unlock Bitcoin native SNARK verification, we need to emulate missing covenants and use a hash-based commitment scheme-enabled proof system to enable the verificaiton without the upgrade.

2.2 Placeholder Proof System

Placeholder [1] is a zero-knowledge succinct non-interactive argument of knowledge based on $\mathcal{PLONK}$-style arithmetization. Placeholder's commitment scheme and types of arithmetization, are replaceable and configurable. Low-level Placeholder circuits can adapt to selected parameters, such as table size, date degree, and lookup options. These properties enable the flexible configuration of Placeholder with trade-offs between circuit parameters, trust assumptions, and efficiency of proof generation. Due to this flexibility, Placeholder can accommodate particular cases, consistently achieving efficient results.

zkSNARK is a type of zero-knowledge proof system that allows one to prove the authenticity of a statement to a verifier without revealing any additional information beyond the statement's validity. The "succinct" and "non-interactive" aspects of zk-SNARKs refer to the fact that the proof is short and does not require any interaction between the prover and verifier beyond the initial setup.

Conceptually, general SNARK construction contains three steps:

1. Translate the problem into a set of polynomials.
2. Commit the polynomials.
3. Prove some relations on the committed polynomials.

Placeholder follows this general SNARK construction and contains two main modules:

1. Arithmetization: Defines the arithmetic representation of the proving statement. Placeholder uses $\mathcal{PLONK}$-based representation with custom gates. The idea was introduced in the TurboPLONK paper [2] and modified later in other proof systems like Halo2 [3] and Kimchi.
2. Commitment Scheme: Placeholder uses the List Polynomial Commitment scheme ([4], [1]) for polynomials obtained from the arithmetization procedure.

Because of the use of List Polynomial Commitment (LPC), Placeholder is positioned as a perfect proof system to be verified on Bitcoin.

But the problem is that even this method doesn't guarantee the so-called "pessimistic" verification on Bitcoin because of commitments-only check being possible. To verify the circuit part, it is required to introduce `OP_MUL` and a larger acceptable script size which requires Bitcoin protocol upgrade. This means a different method should be introduced and it is required to emulate all the following at once:

1. Missing covenants (e.g. CAT)
2. Missing opcodes (e.g. `OP_MUL`)
3. Larger script size

2.3 Functional Encryption

Functional Encryption (FE) is a technique that allows computation over encrypted data to yield decrypted results. It supports restricted secret keys that enable a key holder to learn a specific function of the encrypted data, without learning anything else about the data. For example, given an encrypted program, the secret key may enable the key holder to learn the output of the program on a specific input without learning anything else about the program.

The concept of Functional Encryption was formally studied by Boneh, Sahai, and Waters in [5], who provided precise definitions and discussed its security challenges. The security of FE is non-trivial to define; a natural game-based definition is inadequate for some functionalities, leading to a simulation-based definition, which, while provably secure in the random oracle model, cannot be satisfied in the standard model.

In a functional encryption system, a decryption key allows a user to learn a function of the encrypted data. Briefly, in an FE system for functionality $F(\cdot,\cdot)$ (modeled as a Turing Machine), an authority holding a master secret key can generate a key sk_k that enables the computation of the function $F(k,\cdot)$ on encrypted data. More precisely, using sk_k, the decryptor can compute $F(k,x)$ from an encryption of x. The security of the system guarantees that one cannot learn anything more about x.

An FE scheme for a functionality F is a tuple of four polynomial-time algorithms: *setup*, *keygen*, *enc*, and *dec*, satisfying the following correctness condition for all k in the key space K and x in the plaintext space X:

1. $(pp, mk) \leftarrow setup(1^\lambda)$ (generate a public and master secret key pair)
2. $sk \leftarrow keygen(mk, k)$ (generate secret key for k)
3. $c \leftarrow enc(pp, x)$ (encrypt message x)
4. $y \leftarrow dec(sk, c)$ (use sk to compute $F(k,x)$ from c)

The output y should equal $F(k,x)$ with probability 1.

Standard public-key encryption is a simple example of functional encryption where $K = \{1, \epsilon\}$ and $F(k,x) = x$ if $k = 1$ and $F(k,x) = \text{len}(x)$ if $k = \epsilon$.

2.4 Function-Hiding Inner Product Encryption

Besides many different kinds of FE schemes out there, as reported in [6], in many real scenarios it is important to consider also the privacy of the computed function. The motivation behind this is the fact that a typical workflow of a Bitcoin transaction involves techniques around manipulating pre-signed transactions (or an encrypted signing key) which, if being revealed not in the right moment, would break the whole protocol. If the FE scheme in use does not guarantee any hiding of the function (which is the case for many existing FE schemes), then a hypothetical key sk_f might reveal the predicate functionality contents f, which is undesirable when f itself contains sensitive information (aka a pre-signed transaction or a private key). This has motivated the study of function privacy in FE, see for instance [7, 8, 9].

An IPE scheme is called function-hiding if the keys and ciphertexts reveal no additional information about the related vectors beyond their inner product. The fully function-hiding IPE achieves the most robust IND-based notion of both data and function privacy in the private-key setting in the standard model. The model of full function privacy is sketched hereas described in [9, 10]. Adversaries are allowed to interact with two left-or-right oracles $KeyGen_b(mk,\cdot,\cdot)$ and $Enc_b(mk,\cdot,\cdot)$ for a randomly chosen $b \in \{0,1\}$, where $KeyGen_b$ takes two functions f_0 and f_1 as input and it returns a functional decryption key $sk_{f_b} = KeyGen(mk, f_b)$. The algorithm Enc_b takes two messages x_0 and x_1 as input and it outputs a ciphertext $c_{x_b} = Enc(mk, x_b)$. Adversaries can adaptively interact with oracles for any polynomial (a priori unbounded) number of queries. To exclude inherently inevitable attacks, there is a condition for adversarial queries that all pairs (x_0, x_1) and (f_0, f_1) must satisfy $f_0(x_0) = f_1(x_1)$.

Only two approaches have been proposed for (fully) function-private IPE schemes in the private-key setting. One is to employ the Brakerski-Segev general transformation from (non-function-private) FE schemes for general circuits [8]. The transformation itself is efficient since it simply combines symmetric key encryption with FE in a natural manner. Anyway, this approach requires computationally intensive cryptography tools, such as IND obfuscation, to realise non-function-private FE for general circuits, meaning it may be relatively inefficient overall. The other approach may be more practical. It directly constructs IPE schemes by using the dual-pairing vector spaces (DPVS) introduced by Okamoto and Takashima [11, 12].

In the last few years, there has been a flurry of works on the construction of function-hiding IPE, starting with the work of Bishop, Jain, and Kowalczyk [6]. They propose a function-hiding IPE scheme

under the SXDH assumption, which satisfies an adaptive IND-based security definition. But the security model has one limitation: all ciphertext queries x_0, x_1 and all secret key queries y_0, y_1 are restrained by $\langle x_0, y_0\rangle = \langle x_0, y_1\rangle = \langle x_1, y_0\rangle = \langle x_1, y_1\rangle$. In [9], Datta et al. develop a full function-hiding IPE scheme built in the setting of asymmetric bilinear pairing groups of prime order. The security of the scheme is based on the well-studied SXDH assumption where the restriction on adversaries' queries is only $\langle x_0, y_0\rangle = \langle x_1, y_1\rangle$. Here, secret keys and ciphertexts of n-dimensional vectors consist of $4n + 8$ group elements. Tomida et al., in [13], construct a more efficient function-hiding IPE scheme than that of [9] under the XDLIN assumption, where secret keys and ciphertexts consist of $2n + 5$ group elements. Kim et al., in [14], put forth a fully-secure function-hiding IPE scheme with smaller parameter sizes and run-time complexity than those in [6, 9]. The scheme is proved SIM-based secure in the generic model of bilinear maps. In [15], Zhao et al. present the first SIM-based secure secret-key IPE scheme under the SXDH assumption in the standard model. The authors claim that the scheme can tolerate an unbounded number of ciphertext queries and adaptive key queries. Zhao et al. in [16] propose a new version of the scheme, which is an improvement in terms of computational and storage complexity. In a very recent work [17], Liu et al. present a more efficient and flexible private-key IPE scheme with SIM-based security. To ensure correctness, the scheme requires that the computation of inner products is within a polynomial range, where the discrete logarithm of $g^{\langle x,y\rangle}$ can be found in polynomial time. In [17], the authors compare their proposed IPE scheme with those in [9, 13, 15, 16]. The performance of this scheme appears superior in both storage complexity and computation complexity. Moreover, secret keys and ciphertexts are shorter.

Although most aforementioned IPE schemes are efficient and based on standard assumptions, they all have one inconvenient property: they are bounded. The maximum length of vectors has to be fixed at the beginning, and afterward, one cannot handle vectors whose lengths exceed it. This could be inconvenient when it is hard to predict which data will be encrypted in the setup phase. One may think to solve the problem by setting the maximum length to a large value. However, the size of parameters expands at least linearly with the fixed maximum length, and such a solution incurs an unnecessary efficiency loss. In the context of IP-PE and ABE, there exist unbounded schemes (see, for instance, [18, 19, 20, 6]), whose public parameters do not impose a limit on the maximum length of vectors or number of attributes used in the scheme. In [13], Tomida and Takashima construct two concrete unbounded IPE schemes based on the standard SXDH assumption, both secure in the standard model: the first is a private-key IPE with fully function hiding, the second scheme is a public-key IPE with adaptive security. Concurrently and independently, in [21], Dufour-Sans and Pointcheval describe an unbounded IPE system supporting identity access control with succinct keys. Their construction is proved selectively IND-secure in the random oracle model based on the standard DBDH assumption. In [13], it is shown a comparison, in terms of efficiency, among private-key schemes that are fully function hiding [20, 19, 15] and public-key schemes with adaptive security in the standard model [22].

3 Proposal

To achieve succinct verification of Placeholder (and in general SNARK) proofs on Bitcoin practically two approaches can be considered:

1. Implement trivial covenants (e.g. CAT) with Bitcoin PIPE (FH-MIPE predicates) and to implement LPC/FRI commitments verification using FH-MIPE covenant signing as an equivalent of using `OP_CAT` itself and to eventually introduce `OP_MUL` as an additional opcode.

   This approach would lead to a composite covenant which would allow to verify Placeholder on Bitcoin by verifying commitments in 8 and 6, but not the gates part (aka the circuit definition - "Quotient polynomial check", "Verify Basic Constraints", "Verify Lookup Argument" and "Verify Permutation Argument" in 7) as it would require the emulation of a multiplication opcode. This means the verification would still be some considered some kind of "optimistic" 'cause only commitments will be checked (i.e. $LPCVer$ and $FRIVer$) (no constraint-related checks will be made).

2. Implement the whole ZKP (e.g. Placeholder proof system [1]) verification procedure within the Bitcoin PIPE using some FH-MIPE scheme enabling complex predicates, which would enable a complete Placeholder verification on Bitcoin, but it would be a monolithic application-specific covenant.

Considering the complexity of the Placeholder verifier, both of these verification approaches would require to pick a functional encryption scheme, which would allow predicates to be defined with at least polynomial-size circuits (such schemes can be found among predicate encryption schemes) because to make the functionality to behave like a covenant it would be required to make it to:

1. Encrypt the Schnorr signing key sk.
2. Verify some proof system proof (IPA/KZG-based ([23]) most probably).
3. If the verification is successful, then to sign input user data m with sk during decryption predicate execution.

   This also means the functionality would need to define, among other primitives, Schnorr signing algorithm.

Or, more formally, the overall process would look like this:

Algorithm 1 *CovenantSetup*

Input:
FE master keys: $(m, M) \in K = \{0,1\}_* \cup \{\epsilon\}$,
Schnorr Keys: (sk, pk),
Public Inputs: $input$, $proof$
Transaction: tx

Output: Functionality $C_f : K \times X \Leftrightarrow \{0,1\}_*$

$m: \ C_p \leftarrow Encrypt(M, sk)$
$PrivateKey(sk, proof, input) = Add(m, \mathsf{Verify}(proof, input))$
$Publickey(pk, proof, input) = TweakAdd(pk, \mathsf{Verify}(proof, input))$
$F(sk, tx, proof) =$ **if** $\mathsf{Verify}(proof, input) = \mathsf{true}$ **then return** $SchnorrSign(sk, \ tx)$ **else return** true
Compute the public key: $u \leftarrow PublicKey(pk, \ proof, input)$
Encrypt the function: $C_f = EncryptFunction(m, \ F)$
Protocol initializer erases m and sk.

$$\sigma \leftarrow C_f(C_p, \ Encrypt(M, tx))$$

From now on, σ can be used as a signature of u over tx. If m and sk were erased, C_f can only sign in case the covenant functionality F is "executed" correctly.

3.1 Functional Encryption Schemes for Circuits

To define such functionality, we need an an FE (or PE) scheme for circuits. Functional Encryption schemes for circuits have been studied extensively due to their powerful expressive abilities. Such schemes allow computation of arbitrary circuits over encrypted data, enabling a wide range of applications. The main goal in this area is to construct FE schemes that support circuits of arbitrary polynomial size while maintaining security and efficiency.

In [24], Gorbunov et al. present a leveled PE scheme for all circuits (Boolean predicates of bounded depth), with succinct ciphertexts and secret keys independent of the size of the circuit. The achieved privacy notion is a selective SIM-based variant of attribute-hiding, assuming the hardness of the subexponential LWE problem. Recall that the strong variant notion (full attribute-hiding) is impossible to realize for many messages [25].

Some results have also been obtained for general-purpose FE, in which functions associated with secret keys can be any arbitrary circuits. In [26], Garg et al. give constructions for IND obfuscation (based on multilinear maps), and they use it to construct FE for all polynomial-size circuits. In [27], there are constructions directly based on multilinear maps. Further analysis can be found in [25] where Lin and Tessaro reduce the degree of the required multilinear map to 3.

One of the early works in this area was by Gorbunov, Vaikuntanathan, and Wee in 2015, who proposed a general-purpose FE scheme for circuits based on learning with errors (LWE) [28]. This scheme supports

the evaluation of any polynomial-size circuit on encrypted data while ensuring that the only information leaked is the output of the circuit.

In 2016, Garg, Gentry, Halevi, and Zhandry introduced a more efficient construction based on the notion of multi-input functional encryption (MIFE) [29]. Their scheme supports the evaluation of circuits on inputs encrypted under different keys, which broadens the applicability of FE to scenarios involving multiple data sources.

Recently, there has been a significant focus on improving the efficiency of FE schemes for circuits. A notable contribution in this direction was made by Ananth, Brakerski, and Vaikuntanathan in 2017, who proposed an FE scheme for circuits with better security guarantees and reduced ciphertext size [30].

It is also required to take into consideration that an obfuscated/encrypted signing entity private key is necessary to be baked into the covenant, which means the choice of a particular FE scheme would depend on if it supposes the presence of IO or function-hiding.

Unfortunately none of the existing FE schemes for circuits satisfy the requirements as the requirement to maintain function-privacy introduces necessity for IO, which makes the scheme inefficient. Which means it is necessary to look for a workaround.

3.2 PIPE: Polynomial (Function-Hiding) Inner Product Encryption

As mentioned, it is required to have a Schnorr signing key encrypted to make an FE scheme to behave like a covenant. This means the FE scheme should be a function-hiding one to avoid revealing the key itself. If the FE scheme in use does not guarantee any hiding of the function (which is the case for many existing FE schemes), then the key sk_f might reveal f, which is undesirable when f itself contains sensitive information (Schnorr signing key). As discussed in 2.4, the scheme of a choice should be either:

1. Function-hiding one capable of having predicates deifned with circuits.
2. Multi-input one with indistinguishable obfuscator being present, again, being capable of defining predicates with circuits.

3.2.1 Inner-Product Functional Encryption

To satisfy all the requirements of the concept though, a modification of a DPVS-based scheme described in [31] is proposed to be used. The scheme proposed to be used enables inner-product functionalities to be computed and is designed to be instantinated under pretty widespread SXDH assumptions, but, unfortunately, it is not generic enough as it doesn't support general-purpose circuits, straightforward introduction of which, as discussed in 2.4, would require to introduce indistinguishable obfuscation or encrypt inputs in a way similar to Brakerski-Segev transformation, which would make the scheme impractical.

To bypass the necessity for an IO, it is proposed to replace direct circuit definition (aka using FE for circuits) with embedding linearizable proof system verification and signature generation primitives into the resulting inner product pairing computation. Since the initial scheme is based on DPVS framework, this way it would be possible to keep the Schnorr signing key secret (to not to reveal it to the decryptor) and in the same time it makes it possible to define various computations as a condition for the signing to happen.

3.2.2 Initial Scheme: FH-MIPE

The FH-MIPE scheme to begin with ([32], [31]) consists of four main algorithms: *Setup*, *KeyGen*, *Encryption*, and *Decryption*:

Algorithm 2 FH-MIPE.Setup

procedure FH-MIPE.$\textsc{Setup}(m, n, B, \lambda)$

Generate bilinear group parameters $\mathcal{G} = (q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, g_1, g_2, e) \leftarrow \mathcal{G}_{bpg}(1^\lambda)$ with $q \gg nB$

Generate DPVS parameters $\mathcal{V} = (q, \mathbb{V}_1, \mathbb{V}_2, \mathbb{G}_T, \mathcal{A}_1, \mathcal{A}_2, e) \leftarrow \mathcal{G}_{dpvs}(2m + 2k + 1, \mathcal{G})$

Sample $\nu \in \mathbb{F}_q \setminus \{0\}$, compute $g_T = e(g_1, g_2)^\nu$

for $\iota \in [n]$ **do**

Generate dual orthonormal bases:

$$(\mathcal{B}_\iota = \{\mathbf{b}_{\iota,j}\}_{j=1}^{2m+2k+1}, \mathcal{B}_\iota^* = \{\mathbf{b}_{\iota,j}^*\}_{j=1}^{2m+2k+1}) \leftarrow \mathcal{G}_{ob}(2m + 2k + 1, \mathcal{V}, \nu)$$

end for

Output public parameters:

$$pp = (\mathcal{G}, \mathcal{V}, g_T, \{\mathcal{B}_\iota\}_{\iota\in[n]})$$

Output master secret key:

$$msk = (\nu, \{\mathcal{B}_\iota^*\}_{\iota\in[n]})$$

end procedure

Algorithm 3 FH-MIPE.KeyGen

procedure FH-MIPE.$\textsc{KeyGen}(pp, msk, \{\mathbf{y}_\iota\}_{\iota\in[n]})$

for $\iota \in [n]$ **do**

Sample random scalars $\gamma_{\iota,1}, \gamma_{\iota,2} \leftarrow \mathbb{F}_q$

Compute:

$$\mathbf{sk}_\iota = (\mathbf{y}_\iota, \mathbf{0}_{2m+2k+1}, \gamma_{\iota,1}, \gamma_{\iota,2})$$

end for

Output decryption key:

$$sk = \{\mathbf{sk}_\iota\}_{\iota\in[n]}$$

end procedure

Algorithm 4 FH-MIPE.Encrypt

procedure FH-MIPE.$\textsc{Encrypt}(pp, msk, \iota, \mathbf{x}_\iota)$

Choose random elements $\phi_{\iota,1}, \ldots, \phi_{\iota,k} \xleftarrow{\cup} \mathbb{F}_q$.

Compute

$$\boldsymbol{c}_\iota = \sum_{j\in[m]} x_\iota^{(j)} \boldsymbol{b}_{\iota,j} + \boldsymbol{b}_{\iota,2m+1} + \sum_{j\in[k]} \phi_{\iota,j} \boldsymbol{b}_{\iota,2m+k+j} = (\boldsymbol{x}_\iota, \mathbf{0}_m, 1, \mathbf{0}_{k-1}, \phi_{\iota,1}, \ldots, \phi_{\iota,k}, 0)\, \boldsymbol{B}_\iota,$$

where $\boldsymbol{B}_\iota$ is extracted from msk.

Output the ciphertext $ct_\iota = (\iota, \boldsymbol{c}_\iota)$.

end procedure

Algorithm 5 FH-MIPE.Decrypt

procedure FH-MIPE.$\textsc{Decrypt}(pp, sk, \{\mathbf{ct}_\iota\}_{\iota\in[n]})$

Compute:

$$L_T = \prod_{\iota\in[n]} e(\mathbf{ct}_\iota, \mathbf{sk}_\iota)$$

Find $\Lambda \in \mathbb{Z}$ such that:

$$g_T^\Lambda = L_T$$

Output Λ if successful; else, output $\perp$

end procedure

3.2.3 FH-MIPE to Quadratic Function Encryption (QFE)

In the original FH-MIPE scheme, the decryption computes an inner product between the encrypted vectors $\mathbf{x}_\iota$ and the secret key vectors $\mathbf{y}_\iota$, resulting in a linear function evaluation of the form:

$$\Lambda = \sum_{\iota=1}^{n} \langle \mathbf{x}_\iota, \mathbf{y}_\iota \rangle.$$

However, the goal is to generate a Schnorr signature during decryption to induce covenant-alike behavior, which involves operations that are inherently nonlinear, specifically exponentiation and hashing outputs. Additionally, it is neccesary to integrate the verification of a KZG-based Placeholder proof, which involves verifying polynomial relationships and pairing equations that are quadratic in nature.

Given that both the Schnorr signature generation and the KZG-based Placeholder verification require quadratic computations, we need a functional encryption scheme that supports quadratic functions. The original FH-MIPE scheme is limited to linear functions (inner products) and cannot natively support these quadratic computations within its decryption process.

Therefore, a transition to Quadratic Functional Encryption (QFE) is necessary. QFE schemes allow for the evaluation of quadratic functions over encrypted data during decryption. This enables us to compute the required nonlinear operations and integrate the verification steps directly into the decryption algorithm.

3.2.4 Incompatibility with Inner Product Pairing Computation

The inner product pairing computation in the original FH-MIPE scheme cannot accommodate the generation of a Schnorr signature or the KZG-based Placeholder verification.

- **Schnorr Signature Generation**: The computation of a Schnorr signature involves multiplying the secret signing key x_{sign} with a hash output e, and adding a random nonce k. This operation is of the form:

  $$s = k + x_{\text{sign}} \cdot e \mod q,$$

  which is a nonlinear operation involving both addition and multiplication of secret values. An inner product computation cannot capture this operation, as it is limited to linear combinations.
- **KZG-Based Placeholder Verification**: The verification of KZG commitments involves checking polynomial relationships and pairing equations that are quadratic in the exponents. These verifications require the ability to compute products of encrypted values, which is beyond the capability of inner product computations.
- **Quadratic Equations in Verification**: The verification equations for the Placeholder proof involve terms like $a(z) \cdot b(z)$, which are products of polynomial evaluations at a point z. Representing and verifying such equations require quadratic computations over the encrypted data.

Thus, to achieve the goal, we need a functional encryption scheme capable of handling quadratic computations, which necessitates the transition to Quadratic Functional Encryption.

3.2.5 Transition from MIPE to Quadratic Functional Encryption (QFE)

As established, the FH-MIPE scheme supports multi-input inner product functionality, which is insufficient for the purpose. We need to extend the scheme to support quadratic functions with following modifications:

- **Key Generation**: Modify the key generation algorithm to produce keys corresponding to quadratic functions.
  Given a symmetric matrix $\mathbf{M} \in \mathbb{F}_q^{n \times n}$ defining the quadratic function $f(\mathbf{x}) = \mathbf{x}^\top \mathbf{M} \mathbf{x}$, the functional decryption key is defined as:

  $$sk = (\nu, \mathbf{M}, \{\mathcal{B}_\iota^*\}_{\iota \in [n]}).$$

- **Encryption**: Adjust the encryption algorithm to encrypt the input vector $\mathbf{x}$.
  For an input vector $\mathbf{x} \in \mathbb{F}_q^n$, the encryption algorithm computes ciphertexts:

$$ct_\iota = g_2^{x_\iota}, \quad \text{for } \iota \in [n].$$

- **Decryption**: Modify the decryption algorithm to compute quadratic function evaluations during decryption.
  The decryption algorithm computes:

$$L_T = \prod_{\iota=1}^{n} \prod_{j=1}^{n} e(ct_\iota, ct_j)^{M_{\iota j}} = e(g_2, g_2)^{\mathbf{x}^\top \mathbf{M} \mathbf{x}}$$

  .
  Since $g_T = e(g_1, g_2)^\nu$ and $e(g_2, g_2) = e(g_1^{\nu^{-1}}, g_2)$, we have:

$$L_T = e(g_1, g_2)^{\mathbf{x}^\top \mathbf{M} \mathbf{x}}$$

  .
  Therefore, we can recover $\Lambda = \mathbf{x}^\top \mathbf{M} \mathbf{x}$ such that:

$$g_T^\Lambda = L_T.$$

3.3 Bitcoin PIPE: Incorporate Schnorr Signature Generation

What makes an FH-MIPE-encrypted Schnorr private key a covenant is a predicate which yields a Schnorr signature (R, s) on a message m executed within the QFE-defined predicate. This embedding modifies the process to include following steps:

- **Function Definition**: Define the function $f(k, x_{\text{sign}}) = (R, s)$, where $R = k \cdot G$, $s = k + x_{\text{sign}} \cdot e \mod q$, and $e = H(P \parallel R \parallel m)$.
- **Encryption**:
  1. **Nonce Generation**: Generate nonce k deterministically using a SHA2-256 hash function modeled within QFE inside the predicate, e.g.,

$$k = SHA2 - 256_{\text{QFE}}(x, m),$$

  2. **Compute Commitment**:

$$R = k \cdot G.$$

  3. **Encrypt Inputs**:

$$ct_x = \text{Enc}(x).$$

$$ct_k = \text{Enc}(k).$$

  4. **Form Ciphertext**:

$$ct = (ct_x, ct_k, R_x, m).$$

- **Decryption**:
  1. **Compute Challenge**:

$$e = \text{SHA2-256}(R_x \parallel P_x \parallel m).$$

     Note: e is either computed externally and treated as a constant in the decryption function, either with a QFE-modelled SHA2-256.
  2. **Signing Expressed in QFE**:

$$s = f(ct_x, ct_k) = k + ex \mod q$$

     . The function f computes s using encrypted inputs ct_x, ct_k, and constant e.
  3. **Output Signature**:

$$\sigma = (R_x, s).$$

- **Verification**: The signature $\sigma = (R_x, s)$ can be verified using Bitcoin's standard Schnorr signature verification algorithm:
  1. Reconstruct R from R_x. Since only R_x is known, we need to handle the ambiguity in the y-coordinate. In Bitcoin, this is resolved by encoding the parity of R_y in the signature or by using the quadratic residue of R_y.
  2. Compute challenge $e = \text{SHA-256}(R_x \parallel P_x \parallel m)$.
  3. Verify that: $sG \stackrel{?}{=} R + eP$.

3.3.1 Embedding Secp256k1 Into a Pairing-Friendly Curve

In a QFE scheme, we operate over pairing-friendly elliptic curves (e.g., BLS12-381) that support bilinear pairings necessary for the scheme's functionality. However, Bitcoin's Schnorr signature scheme operates over the secp256k1 curve, which is not pairing-friendly and uses different underlying field parameters.

To integrate Bitcoin-compatible Schnorr signatures within our QFE scheme, we can consider introducing non-native curve arithmetics to perform computations over secp256k1 within the QFE scheme's framework.

Non-native curve arithmetic involves performing arithmetic operations of one elliptic curve (the non-native curve) within the arithmetic framework of another elliptic curve (the native curve). Specifically, we aim to simulate the operations of secp256k1 (non-native curve) within the field and group operations of the pairing-friendly curve used in the QFE scheme (native curve).

To achieve this, we need to:

- Represent elements of the secp256k1 field $\mathbb{F}_p$ within the field $\mathbb{F}_q$ of the QFE scheme.
- Implement the arithmetic operations of $\mathbb{F}_p$ (addition, multiplication, inversion) using the operations available in $\mathbb{F}_q$.
- Simulate the elliptic curve group operations (point addition, scalar multiplication) of secp256k1 within the QFE scheme.

Sec256k1 Curve The secp256k1 curve is defined over the finite field $\mathbb{F}_p$, where:

$$p = 2^{256} - 2^{32} - 977.$$

The curve equation is:

$$E_{\text{secp}} : y^2 = x^3 + 7 \mod p.$$

Pairing-Friendly Curve The pairing-friendly curve used in the QFE scheme (e.g., BLS12-381) is defined over the finite field $\mathbb{F}_q$, where q is a large prime different from p. The curve equation and group operations are defined accordingly.

Representing $\mathbb{F}_p$ Elements in $\mathbb{F}_q$ Since p and q are different primes, elements of $\mathbb{F}_p$ cannot be directly represented as elements of $\mathbb{F}_q$. Instead, we can represent elements of $\mathbb{F}_p$ as vectors of elements in $\mathbb{F}_q$.

Bit Decomposition An element $a \in \mathbb{F}_p$ can be represented by its binary expansion:

$$a = \sum_{i=0}^{255} a_i \cdot 2^i,$$

where $a_i \in \{0, 1\}$.

Each bit a_i can be represented as an element in $\mathbb{F}_q$ (since $\mathbb{F}_q$ can represent the values 0 and 1).

Emulating $\mathbb{F}_p$ Arithmetic in $\mathbb{F}_q$ To perform arithmetic operations over $\mathbb{F}_p$ within $\mathbb{F}_q$, we can use arithmetic circuits that simulate the field operations using the bitwise representation.

Addition Given $a, b \in \mathbb{F}_p$ with bit representations $\{a_i\}$ and $\{b_i\}$, the sum $c = a + b \mod p$ can be computed by:

1. Perform bitwise addition with carry:

$$c_i = a_i \oplus b_i \oplus c_{i-1},$$

   where c_{i-1} is the carry from the previous bit.

2. Enforce constraints to handle carries and modulus p.

Multiplication Multiplication can be performed using algorithms like:

- **Long Multiplication**: Multiply bit representations and sum appropriately.
- **Karatsuba Algorithm**: Optimize multiplication by reducing the number of multiplications required.

Modular Reduction Since operations are modulo p, we need to implement modular reduction algorithms to ensure results remain within $\mathbb{F}_p$.

Elliptic Curve Point Operations Elliptic curve point addition and scalar multiplication over secp256k1 involve field operations in $\mathbb{F}_p$.

Point Addition Given two points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ on E_{secp}, the point addition formulas are:

$$\begin{aligned} \lambda &= \frac{y_2 - y_1}{x_2 - x_1} \mod p, \\ x_3 &= \lambda^2 - x_1 - x_2 \mod p, \\ y_3 &= \lambda(x_1 - x_3) - y_1 \mod p. \end{aligned}$$

These operations involve field addition, subtraction, multiplication, and inversion, all of which must be emulated within $\mathbb{F}_q$.

Scalar Multiplication Scalar multiplication $R = k \cdot G_{\text{secp}}$ is performed using repeated point additions and doublings, following methods like the double-and-add algorithm.

3.3.2 Quadratic Constraints Representation

To be compatible with the QFE scheme, we need to express the operations as quadratic constraints over $\mathbb{F}_q$.

Example: Multiplication Constraint For variables $a, b, c \in \mathbb{F}_q$, enforcing $c = a \cdot b$ can be represented as a quadratic constraint.

Example: Inversion Constraint Inversion can be enforced by the constraint:

$$a \cdot a^{-1} = 1 \mod p.$$

However, inversion is not quadratic. We can represent inversion using a series of multiplication constraints and auxiliary variables.

3.4 Implementing Schnorr Signature Computation

We aim to compute the Schnorr signature components (R, s) within the QFE scheme, where operations are defined over secp256k1.

3.4.1 Signature Components

- **Nonce Generation**: $k \in \mathbb{F}_n$, where $n = p$ (order of secp256k1).
- **Commitment Computation**: $R = k \cdot G_{\text{secp}}$.
- **Challenge Computation**: $e = H(R, m)$.
- **Response Computation**: $s = k + x_{\text{sign}} \cdot e \mod n$.

3.4.2 Implementation Steps

1. **Represent Scalars and Points**

   Represent k, x_{sign}, e, and the point coordinates within $\mathbb{F}_q$ using their bitwise representations.

2. **Emulate Scalar Multiplication**

   Compute $R = k \cdot G_{\text{secp}}$ by emulating scalar multiplication within the constraints of the QFE scheme.

3. **Compute Challenge**

   Since $e = H(R, m)$, where H is a hash function, we need to model H within the QFE scheme or treat e as an external input.

4. **Compute Response**

   Compute $s = k + x_{\text{sign}} \cdot e \mod n$ using emulated field addition and multiplication.

   Since it is linear in k and x. To make it compatible with QFE, we can represent s as a quadratic function by considering linear functions as a special case of quadratic functions with zero quadratic coefficients.

   Define the quadratic function $f(k, x)$:

   $$f(k, x) = s = a_{kk}k^2 + a_{xx}x^2 + a_{kx}kx + b_k k + b_x x + c,$$

   where:

   - $a_{kk} = 0$, $a_{xx} = 0$, $a_{kx} = 0$.
   - $b_k = 1$, $b_x = e$, $c = 0$.

   Thus, the function simplifies to:

   $$s = f(k, x) = b_k k + b_x x = k + xe.$$

5. **Express Constraints**

   All operations must be expressed as quadratic constraints over $\mathbb{F}_q$.

3.4.3 Usage

Architecturally speaking, such a PIPE (FH-MIPE) covenant is a ciphertext with a functionality which contains and encrypted Schnorr signing key hidden within the predicate, which if the covenant is not satisfied, would simply not sign the withdrawal transaction.

This means sending to a PIPE-restricted account would look like this:

1. Retrieve an encrypted opcode binary with covenant e and private key sk.
2. Select the covenant instance parameters i if needed.
3. Tweak sk by e_i.
4. Send funds to e_i.

Accordingly, redeeming from the account would look as:

1. Prepare the state transition with a specific tx and collect all necessary verification data.
2. Pass the data and tx to the encrypted opcode binary.
3. Submit a transaction with a signature generated by a covenant binary.

3.5 Zero-Knowledge Proofs Verification PIPE

As the end goal is to use Bitcoin PIPEs covenants to verify ZKPs on Bitcoin (using Placeholder as an example), let's recall the Placeholder verification procedures and define necessary PIPE primitives for it:

Algorithm 6 LPC.EvalVerify

Input:
proof $\mathcal{P}$,
evaluation points $\{\xi^{(k)}\}_{k=0}^{l-1}$,
roots of Merkle trees $\{\mathsf{root}_k\}_{k=0}^{K-1}$,
transcript

Output: verification result = true/false

1: $\{\boldsymbol{z}^{(0)}, \ldots, \boldsymbol{z}^{(l-1)}, \pi\} = \mathtt{parse}(\mathcal{P})$
2: Interpolate polynomials $U_k(X) = \mathsf{lagrange\_interpolation}(\{\xi_j^{(k)}, \boldsymbol{z}_j^{(k)}\})$ for $0 \le k < l, 0 \le j < |\xi^{(k)}|$
3: Compute $V_k(X) = \prod_{j=0}^{|\xi^{(k)}|-1}(X - \xi_j^{(k)})$
4: **if** $\mathsf{FRI.Verify}(\pi, \{\mathsf{root}_k\}_{k=0}^{K}, \{U_k(X)\}_{k=0}^{l-1}, \{V_k(X)\}_{k=0}^{l-1}, \mathsf{transcript}) = \mathsf{false}$ **then return** false
5: **return** true

Algorithm 7 Verify

Input: $\pi_{\mathsf{Placeholder}}, \mathsf{preprocessed\_data}, \mathsf{transcript}$

Output: $\mathsf{true/false}$

1: Parse proof $\pi_{\mathsf{Placeholder}}$ into:

$\pi_{\mathtt{comm}} = \{ \overline{\mathsf{variable}},\ \overline{\mathsf{V\_polynomials}},\ \overline{\mathsf{lookup^{perm}}},\ \overline{\mathsf{quotient}},\ \overline{\mathsf{fixed}}\}$

$\pi_{\mathtt{eval}}$ is evaluation proofs for

$\mathtt{polynomial\_evaluations} = \{$

$w_i(y), w_i(\zeta^d \cdot y), i = 0, \ldots, \mathsf{N_{wt}} - 1, \quad s_i(y), s_i(\zeta^d \cdot y), i = 0, \ldots, \mathsf{N_{pi}} - 1$

for all corresponding $d \in \mathbf{o}$,

$V^\sigma(y), V^\sigma(\zeta \cdot y), a^{\mathrm{perm}}(y), a^{\mathrm{perm}}(\zeta^{-1} \cdot y), l^{\mathrm{perm}}(y), V_L(y), V_L(\zeta \cdot y),$

$\{T_i(y)\}, i = 0, \ldots, N_T - 1$

$c_i(y), c_i(\zeta^d \cdot y), i = 0, \ldots, \mathsf{N_{cn}} - 1, \quad l_i(y), l_i(\zeta^d \cdot y), i = 0, \ldots, \mathsf{N_{lk}} - 1, \quad q_i(y), q_i(\zeta^d \cdot y), i = 0, \ldots, \mathsf{N_{sl}} - 1$

for all corresponding $d \in \mathbf{o}$,

$q^{\mathrm{last}}(y), q^{\mathrm{pad}}(y), L_0(y)\}$

2: **Verify Permutation Argument:**

3: Denote polynomials included in permutation argument as $f_0, \ldots, f_{\mathsf{N_{perm}}-1}$

4: Get values $\{f_i(y)\}, \{S_i^e(y)\}, \{S_i^\sigma(y)\}, V^\sigma(y), V^\sigma(\zeta \cdot y), L_0(y), q^{\mathrm{last}}(y), q^{\mathrm{pad}}(y)$ from $\pi_{\mathsf{Placeholder}}$

5: Calculate

$$F_0(y), F_1(y), F_2(y) = \mathsf{PermArgumentVerify}(y, V^\sigma(y), V^\sigma(\zeta \cdot y), \{f_i(y)\}, \{S_i^e(y)\}, \{S_i^\sigma(y)\}, q^{\mathrm{pad}}(y), q^{\mathrm{last}}(y), L_0(y), \mathsf{transcript})$$

6: **Verify Lookup Argument:**

7: Denote polynomials included in lookup argument as $a_0, \ldots, a_{\mathsf{N_{lk}}-1}$

8: Get values $\{a_i(y)\}, \{l_i(y)\}, a^{\mathrm{perm}}(y), a^{\mathrm{perm}}(\zeta^{-1} \cdot y), l^{\mathrm{perm}}(y), V_L(y), V_L(\zeta \cdot y), L_0(y), q^{\mathrm{last}}(y), q^{\mathrm{pad}}(y)$ from $\pi_{\mathsf{Placeholder}}$

9: Calculate:

$$F_3(y), F_4(y), F_5(y), F_6(y), F_7(y) = \mathsf{LookupArgumentVerify}(\overline{\mathsf{lookup^{perm}}}, \{a_i(y)\}, \{l_i(y)\}, a^{\mathrm{perm}}(y), a^{\mathrm{perm}}(\zeta^{-1} \cdot y), l^{\mathrm{perm}}(y), V_L(y), V_L(\zeta \cdot y), L_0(y), q^{\mathrm{last}}(y), q^{\mathrm{pad}}(y), \mathsf{transcript})$$

10: $\mathsf{transcript}.append(\overline{\mathsf{V\_polynomials}})$

11: **Verify Basic Constraints:**

12: For $i = 0, \ldots, \mathsf{N_{sl}} - 1$

$$g_i(X) = q_i(X) \cdot (\theta^{k_i - 1 + \nu_i} \mathcal{C}_{i_0}(X) + \cdots + \theta^{\nu_i} \mathcal{C}_{i_{k-1}}(X))$$

13: Calculate a constraints-related numerator of the quotient polynomial $F_8(y) = \sum_{0 \le i < \mathsf{N_{sl}}} (g_i(y))$

14: **Quotient polynomial check:**

15: **if** $\sum_{i=0}^{8} \alpha_i F_i(y) \neq Z(y)T(y)$ **then return** $false$

16: Get challenges $\{\alpha_i \in \mathbb{F}\}_{i=0}^8,\ \theta \in \mathbb{F},\ y \in \mathbb{F} \setminus H$ from $\mathsf{transcript}$

17: $\mathsf{transcript}.append(\overline{\mathsf{quotient}})$

18: **Evaluation proof check**

19: **if** $\mathsf{LPC.EvalVerify}(\mathtt{polynomials\_evaluations}, \pi_{comm}, \mathsf{transcript}) = false$ **then return** $false$

Algorithm 8 FRI.Verify

Input: FRI proof π, Merkle roots $\{\mathsf{T\_root}_k\}_{k=0}^{K-1}$, $\{U_k(X)\}_{k=0}^{l-1}, \{V_k(X)\}_{k=0}^{l-1}, \mathsf{transcript}$
Output: verification result = true/false

$\{\mathsf{fri\_root}_0, \ldots, \mathsf{fri\_root}_{\mathsf{steps_{FRI}}-1}, \pi^{(0)}, \ldots \pi^{(r_q-1)}, \mathsf{final\_polynomial}\} = \mathtt{parse}(\pi)$
$\mathcal{D}^{(0)} = \mathcal{D}, \quad \mathcal{D}^{(i+1)} = q_{r_i}\left(\mathcal{D}^{(i)}\right)$, for $i = 0, \ldots, \mathsf{steps_{FRI}} - 2$
for all $k = 0, \ldots, L-1$ **do**
    $\mathsf{transcript}.append(\mathsf{T\_root}_k)$
end for
$\tau := \mathsf{transcript}.challenge(\mathbb{F})$
$t := 0$
for all $i := 0, \ldots, \mathsf{steps_{FRI}} - 1$ **do**
    $\mathsf{transcript}.append(\mathsf{fri\_root}_i)$
    for all $step := 0, \ldots, r_i - 1$ **do**
        $\alpha_t := \mathsf{transcript}.challenge(\mathbb{F})$
        $t := t + 1$
    end for
end for
for all $query = 0, \ldots, r_q - 1$ **do**
    $\{\pi^*, \pi_0, \ldots, \pi_{\mathsf{steps_{FRI}}-1}\} = \mathtt{parse}(\pi^{(round)})$
    $x^{(0)} = \mathsf{transcript}.challenge(\mathcal{D}_0)$
    $x^{(i+1)} = q_{r_i}(x^{(i)}),\ i = 0, \ldots, \mathsf{steps_{FRI}} - 1$
    Construct cosets $S^{(i)} = \{s \in \mathcal{D}^{(i)} \mid q_{r_i}(s) = x^{(i+1)}\}$, for $i = 0, \ldots, \mathsf{steps_{FRI}} - 1$ ▷ $|S^{(i)}| = m^{r_i}$
    `Initial proof check`
    t := 0;
    for all $k := 0, \ldots, K-1$ **do**
        if $\pi^*.\mathsf{auth}_k.\mathsf{root} \neq \mathsf{T\_root}_k$ **then return** $false$
        if $\mathsf{MT.Validate}(\pi^*.\mathsf{auth}_k, \{\pi^*.\mathsf{val}^{(t)}, \ldots, \pi^*.\mathsf{val}^{(t+l_k-1)}\}) = false$ **then return** $false$
        $t := t + l_k$
    end for
    Compute values of combined polynomial Q values val from $\pi_k.\mathsf{val}$

$$\mathsf{val} = \left\{ \prod_{k=0}^{l-1} \tau^{l-k-1} \frac{\pi^*.\mathsf{val}_s^{(k)} - U_k(s)}{V_k(s)} \right\}_{s \in S^{(0)}}$$

    `Round proofs check`
    $t := 0, S := S^{(0)}$
    for all $i := 0, \ldots, \mathsf{steps_{FRI}} - 1$ **do**
        if $\pi_i.\mathsf{auth.root} \neq \pi.\mathsf{fri\_root}_i$ **then return** $false$
        if $\mathsf{MT.Validate}(\pi_i.\mathsf{auth}, \mathsf{val}) = false$ **then return** $false$
        for all $step := 0, \ldots, r_i - 1$ **do**
            $S_{\mathsf{next}} := \{q(s)\}_{s \in S}$
            $\mathsf{interpolant}_s := \mathsf{lagrange\_interpolation}(\{s_j, \mathsf{val}_{s_j}\}_{q(s_j)=s})(\alpha_t)$ for $s \in S_{\mathsf{next}}$
            $t := t + 1, S := S_{\mathsf{next}}, \mathsf{val} := \{\mathsf{interpolant}_s\}_{s \in S_{\mathsf{next}}}$ ▷ $|S_{\mathsf{next}}| = r_i - step - 1$
        end for
        if $\mathsf{val} \neq \pi_i.\boldsymbol{y}_{(x^{(i+1)})}$ **then return** $false$ ▷ $|\mathsf{val}| = 1$
        $\mathsf{val} := \pi_i.\boldsymbol{y}$
    end for
    if $\mathsf{final\_polynomial}(x^{(\mathsf{steps_{FRI}})}) \neq \mathsf{val}$ **then return** $false$
end for
return true

As mentioned in the beginning of this section (3, there are two ways to verify zero-knowledge proofs on Bitcoin. Let's refrain them knowing that now there is an FE scheme capable of complex computations over the ciphertext while maintainng function-privacy (PIPE):

1. Implement trivial covenants (e.g. CAT) with Bitcoin PIPEs (FH-MIPE predicates) and to implement LPC/FRI commitments verification using FH-MIPE covenant signing as an equivalent of using `OP_CAT` itself and to eventually introduce `OP_MUL` as an additional PIPE.
2. Implement the whole ZKP (e.g Placeholder) verification procedure within the Bitcoin PIPE.

Both of these approaches would require for the verification to be converted into the QFE-compatible equations, which means it would preferrable to replace the hash-based commitment scheme with a pairing-based one to be able to fit that easier. The most obvious candidate is KZG ([23]).

3.5.1 Integrate KZG-Based Placeholder Proof Verification

We need to embed the verification of a KZG-based Placeholder proof into the decryption process, ensuring that the Schnorr signature is generated only if the proof verifies correctly. For ths we need to:

- **Include Proof in Ciphertext**: Extend the ciphertext to include the Placeholder proof π. The ciphertext becomes:
$$ct = (ct_1, ct_2, \pi).$$
- **Verification during Decryption**: Modify the decryption algorithm to perform the Placeholder proof verification. During decryption, the decryptor:
  1. Parses the proof π.
  2. Verifies the KZG commitments and the correctness of the polynomial evaluations as per the Placeholder proof system.
  3. Checks the verification equations, which involve pairing equations and polynomial relationships.
- **Express Verification Equations**: Represent the Placeholder verification equations as quadratic equations compatible with the QFE scheme.

3.5.2 KZG Commitments

A KZG commitment to a polynomial $f(X)$ of degree d is computed as:

$$C_f = f(\tau) \cdot G = \sum_{i=0}^{d} f_i \tau^i G,$$

where:

- $\tau \in \mathbb{F}_q$ is a secret.
- G is a generator of $\mathbb{G}_1$.
- f_i are the coefficients of the polynomial $f(X)$.

3.5.3 Verification Primitives

The KZG verification involves checking that a claimed evaluation $f(z)$ of the polynomial $f(X)$ at a point $z \in \mathbb{F}_q$ is correct, given the commitment C_f and a proof π_f. The verification equation is:

$$e(C_f - f(z)G, g_2) = e(\pi_f, g_2^{\tau - z}).$$

Since $g_2^{\tau - z}$ is not directly available, it can be computed as:

$$g_2^{\tau - z} = \frac{g_2^{\tau}}{g_2^{z}},$$

where g_2^{τ} is part of the Structured Reference String (SRS) and g_2^{z} can be computed by the verifier.

3.5.4 Translation to QFE-Compatible Expressions

To integrate these verification steps into the QFE scheme, we need to represent the pairing-based verification equations as quadratic functions over $\mathbb{F}_q$ that can be evaluated within the decryption algorithm.

Representation of Pairing Equations Consider the verification equation:

$$e(C_f - f(z)G, g_2) = e(\pi_f, g_2^{\tau - z}).$$

Taking logarithms in the exponents (since we are working over $\mathbb{F}_q$), we can represent the pairing equation in terms of the exponents:

$$\langle C_f - f(z)G, g_2 \rangle = \langle \pi_f, g_2^{\tau - z} \rangle,$$

where $\langle \cdot, \cdot \rangle$ denotes the pairing in the exponent space.

However, directly computing logarithms in the exponents is not practical due to the Discrete Logarithm Problem. Instead, we can express the pairing equation as a zero equation suitable for QFE:

$$e(C_f - f(z)G, g_2) \cdot e(\pi_f, g_2^{z-\tau}) = 1_{\mathbb{G}_T}.$$

Since $e(A, B) \cdot e(C, D) = e(A \cdot C, B \cdot D)$, we can write:

$$e\left((C_f - f(z)G) \cdot \pi_f, g_2^{z-\tau} \cdot g_2\right) = e(1_{\mathbb{G}_1}, 1_{\mathbb{G}_2}).$$

But this still involves pairing computations.

Quadratic Expression over $\mathbb{F}_q$ To make the verification compatible with QFE, we represent the verification condition as a quadratic equation over $\mathbb{F}_q$ involving the exponents.

Let us denote:

$$C_f = [c_f]G, \quad \text{where } c_f = f(\tau),$$
$$\pi_f = [w_f]G, \quad \text{where } w_f = \frac{c_f - f(z)}{\tau - z}.$$

The pairing equation becomes:

$$e\left([c_f - f(z)]G, g_2\right) = e\left(w_f G, [\tau - z]g_2\right).$$

Since $e(aG, bg_2) = e(G, g_2)^{ab}$, we can write:

$$e(G, g_2)^{(c_f - f(z)) \cdot 1} = e(G, g_2)^{w_f \cdot (\tau - z)}.$$

Thus, the verification condition is:

$$(c_f - f(z)) = w_f(\tau - z).$$

This is an equation over $\mathbb{F}_q$, involving products of values. Since τ is known to the decryptor (as part of msk), the decryptor can compute both sides of the equation and verify whether it holds.

Expressing in Quadratic Form The verification equation:

$$(c_f - f(z)) - w_f(\tau - z) = 0$$

can be represented as a quadratic function in the variables:

$$f_{\text{verify}}(c_f, f(z), w_f, \tau, z) = (c_f - f(z)) - w_f(\tau - z).$$

This equation is linear in c_f, $f(z)$, w_f, and τ, but since w_f depends on c_f, $f(z)$, and τ, the overall relationship is quadratic.

3.5.5 Inclusion in QFE Decryption

During decryption, the decryptor performs the following steps:

1. **Compute c_f and w_f**

   Using the commitments and proofs in π, the decryptor extracts the exponents c_f and w_f (since the decryptor knows τ and can compute discrete logs in $\mathbb{G}_1$).

2. **Verify the Quadratic Equation**

   Evaluate the quadratic function:

   $$f_{\text{verify}} = (c_f - f(z)) - w_f(\tau - z)$$

   and check whether $f_{\text{verify}} = 0$.

3. **Compute Verification Indicator δ**

   Define:

   $$\delta = \begin{cases} 1, & \text{if } f_{\text{verify}} = 0 \\ 0, & \text{otherwise} \end{cases}.$$

4. **Proceed with Decryption** Use δ in the computation of s' as previously described.

3.5.6 Other Verification Equations

Similarly, other verification equations in the KZG-based Placeholder proof system can be translated into quadratic expressions over $\mathbb{F}_q$.

Polynomial Identity Verification For example, verifying that:

$$a(z) \cdot b(z) - c(z) = 0$$

is inherently a quadratic equation in $a(z)$ and $b(z)$. The decryptor can compute $a(z)$, $b(z)$, and $c(z)$ from the evaluations provided in π (since the decryptor can compute discrete logs with knowledge of τ). The verification is then straightforward.

Permutation Argument Verification In the permutation argument, we verify that:

$$Z_{\text{perm}}(z)(s_1(z) + \beta s_2(z) + \gamma) = Z_{\text{perm}}(\omega z)(a(z) + \beta b(z) + \gamma)$$

This equation can be rearranged and represented as a quadratic function over $\mathbb{F}_q$.

3.5.7 Resulting Scheme

Algorithm 9 PIPE.Setup

procedure PIPE.Setup(1^λ)
  Generate cyclic groups $\mathbb{G}_1$, $\mathbb{G}_2$, $\mathbb{G}_T$ of order q.
  Choose generators $G \in \mathbb{G}_1$, $g_2 \in \mathbb{G}_2$.
  Define bilinear pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$.
  Placeholder Setup (KZG Commitments):

- Sample $\tau \in \mathbb{F}_q$.
- Compute $\text{SRS}_{\mathbb{G}_1} = \{\tau^i G\}_{i=0}^{d}$.
- Compute $\text{SRS}_{\mathbb{G}_2} = \{\tau^i g_2\}_{i=0}^{d}$.

  Functional Encryption Setup:

- Sample $msk = (\nu, \tau)$, where $\nu \in \mathbb{F}_q^*$.
- Compute $g_T = e(G, g_2)^\nu$.
- Compute $g_2' = g_2^{\nu^{-1}}$.

  Output $pp = (G, g_2, e, H, \text{SRS}_{\mathbb{G}_1}, \text{SRS}_{\mathbb{G}_2}, g_T, g_2')$.
  Output msk.
end procedure

Algorithm 10 PIPE.KeyGen

procedure PIPE.KeyGen(pp, msk, f)
  Define $f(k, x_{\text{sign}}, \pi_k, \pi_{x_{\text{sign}}}) = s$, where:

- $s = (k + x_{\text{sign}} \cdot e) \cdot v \mod q$.
- $v = \text{Verify}(\pi_k, \pi_{x_{\text{sign}}})$ outputs 1 if proofs are valid, 0 otherwise.

  Generate sk_f for function f using msk.
  Output sk_f.
end procedure

Algorithm 11 PIPE.Encrypt

procedure PIPE.Encrypt(pp, x_{sign}, m)
  Random Nonce Generation:
  Generate random $k \in \mathbb{F}_q$.
  Compute R:
  $R = k \cdot G$.
  Compute Hash e:
  $e = H(P \parallel R \parallel m)$.
  Encrypt k and x_{sign}:
  $ct_k = \text{Enc}(k)$.
  $ct_{x_{\text{sign}}} = \text{Enc}(x_{\text{sign}})$.
  Generate KZG Commitments and Proofs:
  Commit to k and x_{sign}:
  $C_k = \text{Commit}(k)$.
  $C_{x_{\text{sign}}} = \text{Commit}(x_{\text{sign}})$.
  Generate proofs π_k, $\pi_{x_{\text{sign}}}$.
  Form Ciphertext:
  $ct = (ct_k, ct_{x_{\text{sign}}}, R, C_k, C_{x_{\text{sign}}}, \pi_k, \pi_{x_{\text{sign}}})$.
end procedure

Algorithm 12 PIPE.Decrypt

procedure PIPE.DECRYPT(pp, sk_f, ct, m)
  Parse $ct = (ct_k, ct_{x_{\text{sign}}}, R, C_k, C_{x_{\text{sign}}}, \pi_k, \pi_{x_{\text{sign}}})$.
  Compute Hash e:
  $e = H(P \parallel R \parallel m)$.
  Compute s Using Functional Decryption:
  $s = \text{Dec}(sk_f, ct_k, ct_{x_{\text{sign}}}, \pi_k, \pi_{x_{\text{sign}}}, e)$.
  The decryption function internally:

- **Performs Placeholder Verification:**
- $v = \text{Verify}(\pi_k, \pi_{x_{\text{sign}}})$.
- **Computes s if $v = 1$:**
- $s = (k + x_{\text{sign}} \cdot e) \mod q$.
- **Outputs s if verification succeeds; otherwise, fails.**

  Output Signature:
  The signature is (R, s).
end procedure

3.5.8 Verification of the Schnorr Signature

After decryption, if $\delta = 1$, the signature (R, s') is valid. The verifier checks:

Algorithm 13 PIPE.Verify

procedure PIPE.VERIFY(P, R, s, m)
  Compute Hash e:
  $e = H(P \parallel R \parallel m)$.
  Verify Signature Equation:
  Check if $s \cdot G = R + e \cdot P$.
  Accept or Reject:
  If the equation holds, accept; else, reject.
end procedure

3.6 Verification of Zero-Knowledge Proofs on Bitcoin with PIPEs

3.6.1 Verifying FRI/LPC-based Proof Systems (e.g. Placeholder) via *CAT* PIPE

Embedding IPA-based proof system verification into DPVS-based FH-MIPE scheme enables the execution of complex computations over encrypted data while preserving function privacy (the approach leverages the compatibility between quadratic equations-reduceable KZG verification and the inner product computations supported by the FH-MIPE scheme), as an example of a FH-MIPE covenant, for the sake of the Placeholder verification goal, *CAT* was chosen to be expressed via function-hiding functional encryption. This would require to define, besides Schnorr signing procedure, an implementation of a circuit-expressed `OP_CAT` as a predicate. As such an implementation is done, it is possible to split the Verify 7 to several steps and to replace Merkle-tree verification-related operations of the hash-based commitment scheme with the *CAT* PIPE transaction expectancy. This way an optimistic verification of commitment schemes can be achieved.

3.6.2 Placeholder Verification as a Covenant

As the circuit which defines predicate within the chosen FH-MIPE scheme is possible to be arranged to represent a Turing-complete computation, a much more complicated covenant can be constructed. For example, the signing procedure can be prefixed/conditioned with the Placeholder verification procedure. Considering that the resulting scheme supposes for the KZG-based Placeholder verificaton to happen natively, one can reuse the mechanism in case Placeholder is being used directly. In case other proof systems are being used, the recursion into Placeholder would be enough to embed the verification of those into Bitcoin PIPEs framework.

4 Conclusion

This paper proves that a function-hiding (DPVS-based quadrafic functional encryption (QFE) is enough to introduce covenants for Bitcoin without any necessity for a protocol upgrade and express complex application-specific via those by conditionally signing Bitcoin transaction with Schnorr signature if an IPA-based proof system verification (e.g. KZG-based Placeholder) completes successfully.

Having covenants implemented as Bitcoin PIPEs means:

1. Computation integrity verification happening outside of Bitcoin Script, while still being checked by it.
2. Non-interactive protocol definition means no infrastructure is necessary to maintain liveness, which improves security assumptions to native Bitcoin L1-level ones.
3. Setup procedure, due to the nature of the PIPE scheme, supposes the initial covenant deployer to behave honest. This can be improved with traditional DKG (MPC-based trusted setup), so only one participant out of N generating master key would be required to be honest.
4. Trivial covenants remain bytewise and verification wise more efficient.

In the same time, having Placeholder verification implemented as a Bitcoin PIPE covenant brings to the table:

1. Full proof verification, which means not only commitments are being verified (like it is with an `OP_CAT` case), but the verification also includes a circuit (gates) part into it which means security assumptions improve to the strongest ones (no challenger oracle is required to be present within the verification protocol).
2. Single-round ZKP verification on Bitcoin without any trust assumptions other than a trusted setup assumption.

References

1. Cherniaeva A., Shirobokov I., Komarov M. Placeholder. https://www.allocin.it/uploads/placeholder.pdf. 2024.

2. Gabizon A., Williamson Z. J. Proposal: The Turbo-PLONK program syntax for specifying SNARK programs. https://docs.zkproof.org/pages/standards/accepted-workshop3/proposal-turbo_plonk.pdf.

3. PLONKish Arithmetization - The halo2 book. https://zcash.github.io/halo2/concepts/arithmetization.html.

4. Kattis A., Panarin K., Vlasov A. RedShift: Transparent SNARKs from List Polynomial Commitment IOPs. Cryptology ePrint Archive, Report 2019/1400. 2019. https://ia.cr/2019/1400.

5. Boneh D., Sahai A., Waters B. Functional Encryption: Definitions and Challenges. Cryptology ePrint Archive, Paper 2010/543. 2010. https://eprint.iacr.org/2010/543. URL: https://eprint.iacr.org/2010/543.

6. Bishop A., Jain A., Kowalczyk L. Function-hiding inner product encryption // International Conference on the Theory and Application of Cryptology and Information Security / Springer. 2015. P. 470–491.

7. Boneh D., Raghunathan A., Segev G. Function-private identity-based encryption: Hiding the function in functional encryption // Annual Cryptology Conference / Springer. 2013. P. 461–478.

8. Brakerski Z., Segev G. Function-private functional encryption in the private-key setting // Journal of Cryptology. 2018. Vol. 31, no. 1. P. 202–225.

9. Datta P., Dutta R., Mukhopadhyay S. Functional encryption for inner product with full function privacy // Public-Key Cryptography–PKC 2016 / Springer. 2016. P. 164–195.

10. Datta P., Dutta R., Mukhopadhyay S. Strongly full-hiding inner product encryption // Theoretical Computer Science. 2017. Vol. 667. P. 16–50.

11. Okamoto T., Takashima K. Homomorphic encryption and signatures from vector decomposition // International Conference on Pairing-Based Cryptography / Springer. 2008. P. 57–74.

12. Okamoto T., Takashima K. Hierarchical predicate encryption for inner-products // International Conference on the Theory and Application of Cryptology and Information Security / Springer. 2009. P. 214–231.

13. Tomida K., Takashima K. Function-hiding inner product encryption with flexible public parameters // International Conference on the Theory and Application of Cryptology and Information Security / Springer. 2017. P. 312–341.

14. Kim S., Kim J., Seo J.-H. A new approach to practical function-private inner product encryption // Theoretical Computer Science. 2019. Vol. 783. P. 22–40.

15. Simulation-based security of function-hiding inner product encryption / Q. Zhao, Q. Zeng, X. Liu et al. // Science China Information Sciences. 2018. Vol. 61, no. 4. P. 1–3.

16. Zhao Q., Zeng Q., Liu X. Improved construction for inner product functional encryption // Security and Communication Networks. 2018. Vol. 2018.

17. Liu J. et al. A more efficient and flexible private-key IPE scheme with SIM-based security // International Conference on Security and Cryptography for Networks / Springer. 2020. P. 544–562.

18. Decentralizing inner-product functional encryption / M. Abdalla, D. Catalano, D. Fiore et al. // Annual International Conference on the Theory and Applications of Cryptographic Techniques / Springer. 2019. P. 128–157.

19. Chen K., Wee H., Wichs D. Efficient and strongly secure functional encryption: Point and inner product functionalities // Journal of Cryptology. 2018. Vol. 31, no. 1. P. 208–250.

20. Simple functional encryption schemes for inner products / M. Abdalla, F. Bourse, A. Caro et al. // International Conference on the Theory and Application of Cryptology and Information Security / Springer. 2019. P. 30–60.

21. Dufour-Sans Q., Pointcheval D. Unbounded inner-product functional encryption with succinct keys and its application to attribute-based encryption // International Conference on Security and Cryptography for Networks / Springer. 2019. P. 308–329.

22. Abdalla M., Catalano D., Fiore D. Multi-input functional encryption for inner products: Function-hiding realizations and constructions without pairings // Annual International Conference on the Theory and Applications of Cryptographic Techniques / Springer. 2015. P. 597–627.

23. vv ll jj ff, . Constant-Size Commitments to Polynomials and Their Applications // Hutchison, David and Kanade, Takeo and Kittler, Josef and Kleinberg, Jon M. and Mattern, Friedemann and Mitchell, John C. and Naor, Moni and Nierstrasz, Oscar and Pandu Rangan, C. and Steffen, Bernhard and Sudan, Madhu and Terzopoulos, Demetri and Tygar, Doug and Vardi, Moshe Y. and Weikum, Gerhard and Abe, Masayuki. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. Vol. 6477. P. 177–194. Series Title: Lecture Notes in Computer Science.

24. Gorbunov S., Vaikuntanathan V., Wee H. Leveled fully homomorphic signatures from standard lattices // Cryptology ePrint Archive. 2013. https://eprint.iacr.org/2013/142.

25. Sahai A., Seyalioglu H. Worry-Free Encryption: Functional Encryption with Public Keys // Cryptology ePrint Archive. 2010. https://eprint.iacr.org/2010/229.

26. Candidate indistinguishability obfuscation and functional encryption for all circuits / S. Garg, C. Gentry, S. Halevi et al. // 54th Annual Symposium on Foundations of Computer Science (FOCS). 2013. P. 40–49.

27. Lin H., Tessaro S. Indistinguishability Obfuscation from Trilinear Maps and Block-Wise Local PRGs // Advances in Cryptology – CRYPTO 2017. Vol. 10401 of *Lecture Notes in Computer Science*. 2017. P. 530–569.

28. Gorbunov S., Vaikuntanathan V., Wee H. Predicate Encryption for Circuits from LWE // Annual Cryptology Conference / Springer. 2015. P. 503–523.

29. Functional Encryption Without Obfuscation / S. Garg, C. Gentry, S. Halevi et al. // Theory of Cryptography Conference / Springer. 2016. P. 480–511.

30. Ananth P., Brakerski Z., Vaikuntanathan V. Projective arithmetic functional encryption and indistinguishability obfuscation from degree-5 multilinear maps // Annual International Conference on the Theory and Applications of Cryptographic Techniques / Springer. 2017. P. 152–181.

31. Behera B. C., Ramanna S. C. Multi-Input Functional Encryption for Unbounded Inner Products. Cryptology ePrint Archive, Paper 2024/919. 2024. URL: https://eprint.iacr.org/2024/919.

32. Datta P., Okamoto T., Tomida J. Full-Hiding (Unbounded) Multi-Input Inner Product Functional Encryption from the k-Linear Assumption. 2018.

33. Gabizon A., Williamson Z. J., Ciobotaru O. PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge. Cryptology ePrint Archive, Report 2019/953. 2019. https://ia.cr/2019/953.

34. Fast Reed-Solomon interactive oracle proofs of proximity / E. Ben-Sasson, I. Bentov, Y. Horesh et al. // 45th international colloquium on automata, languages, and programming (icalp 2018) / Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2018.

35. Gabizon A., Williamson Z. J. plookup: A simplified polynomial protocol for lookup tables. Cryptology ePrint Archive, Report 2020/315. 2020. https://ia.cr/2020/315.

36. Lookup argument - The halo2 book. https://zcash.github.io/halo2/design/proving-system/lookup.html.

37. Chiesa A., Ojha D., Spooner N. Fractal: Post-Quantum and Transparent Recursive Proofs from Holography. Cryptology ePrint Archive, Report 2019/1076. 2019. https://ia.cr/2019/1076.

38. Function-hiding inner product encryption is practical / S. Kim, K. Lewi, A. Mandal et al. // International Conference on Security and Cryptography for Networks / Springer. 2018. P. 544–562.

39. Shor P. W. Algorithms for quantum computation: Discrete logarithms and factoring // Proceedings of the 35th Annual Symposium on Foundations of Computer Science. 1994. P. 124–134.

40. Debnath S. R., Patranabis D. MQ-based functional encryption for inner products // International Conference on the Theory and Application of Cryptology and Information Security / Springer. 2020. P. 171–202.

41. Cocks C. An identity-based encryption scheme based on quadratic residues // Proceedings of the 8th IMA International Conference on Cryptography and Coding. 2001. P. 360–363.

42. Agrawal S., Goyal R., Tomida J. Multi-Input Quadratic Functional Encryption: Stronger Security, Broader Functionality // Theory of Cryptography / Ed. by E. Kiltz, V. Vaikuntanathan. Cham: Springer Nature Switzerland, 2022. P. 711–740.

43. Datta P., Okamoto T., Takashima K. Efficient Attribute-Based Signatures for Unbounded Arithmetic Branching Programs. Cryptology ePrint Archive, Paper 2019/363. 2019. URL: https://eprint.iacr.org/2019/363.

44. Tomida J., Abe M., Okamoto T. Efficient Functional Encryption for Inner-Product Values with Full-Hiding Security // Information Security / Ed. by M. Bishop, A. C. A. Nascimento. Cham: Springer International Publishing, 2016. P. 408–425.

45. Groth J. On the Size of Pairing-based Non-interactive Arguments. Cryptology ePrint Archive, Paper 2016/260. 2016. URL: https://eprint.iacr.org/2016/260.