Un-FE'd Covenants: Char-ting a new path to Emulated Covenants via BitVM Integrity Checks

Jeremy Rubin*

November 26, 2024

Abstract

Covenants in Bitcoin represent a method to restrict how and where coins can move. Functional Encryption (FE) offers an exciting avenue to implement covenants without native protocol changes. However, FE remains impractical with current cryptographic tools. In this work, we propose a practical implementation using an oracle-assisted model that combines off-chain computation, key management, and a BitVM-style economic incentive structure to enforce covenants without requiring a Bitcoin soft fork.

1. Introduction

The difficulty in upgrading Bitcoin for smart contracts has inspired innovative approaches to extending its functionality. **FE'd Up Covenants**[Rub24] leverage Functional Encryption (FE) for off-chain enforcement of spending conditions. **Bitcoin PIPE**[Kom24] builds on this with a practical implementation, while **ColliderScript**[HKLP24] offers advanced contract composition via equality checks based on computing 2-way 160-bit hash collisions[1], though at high computational cost. Finally, **BitVM**[Lin23] introduces fraud-proof mechanisms to verify off-chain logic on-chain. These frameworks, each with trade-offs, drive Bitcoin's programmability. This paper advances these ideas, presenting a hybrid approach to implement covenants using cryptographic primitives and economic incentives enforced by a BitVM style penalty bond.

Shameless plug: This work is funded by Char, a new protocol designed to support the growing Bitcoin Layer Two and BitVM ecosystem. If you are inspired by this research and development, please reach out at the email below – we are actively hiring motivated individuals to join our team. Or, to just keep in touch, join our discussion group.

*jeremy [dot] L [dot] rubin [at] gmail [dot] com

[1]leveraging the hardness gap between 2-way and 3-way hash collisions

1.1. Why FE Isn't Ready Yet

Functional Encryption (FE), while theoretically powerful, remains elusive for practical deployment. The cryptographic primitives required for secure and efficient FE have not reached maturity, leaving the direct implementation of FE-based covenants out of reach for now. While work such as Bitcoin PIPE is promising, it remains to be seen if the computational costs will be excessive or if the security model (and hardness) will be sufficient.

1.2. Motivation for an Oracle-Assisted Approach

Given the current state of FE, we propose a hybrid approach leveraging existing cryptography and a novel economic incentive structure. By using an oracle server to run the covenant logic and cryptographic key-tweaking, we can emulate many of the benefits of FE-based covenants today. Additionally, we introduce a BitVM-style incentive mechanism to ensure the oracle adheres to the covenant logic.

1.3. Overview of This Paper

This paper details:

(I) The architecture for an oracle-assisted covenant system.
(II) How the oracle enforces covenants by running the program logic and managing keys.
(III) An economic incentive structure using a BitVM fraud-proof mechanism to hold the oracle accountable.

2. Oracle-Assisted Covenants

2.1. Architecture

2.1.1. Trusted Oracle Execution

The oracle serves as a trusted intermediary. It generates and manages keys and runs the covenant program logic. Specifically:

(I) Generate a SECP256K1 Keypair: $(k, \mathcal{K}) \longleftarrow KeyGen()$.
(II) Compute program-specific tweaks for each covenant instance [2]:

$$UniquePubKey(\mathcal{K}, E_I) = Derive(\mathcal{K}, Sha256(E_I)).$$

(III) Execute program logic securely within the oracle's environment and sign conditionally.

[2] Denoted E for "encumbrance". We differentiate a covenant primitive, a covenant, and a covenant instance. A covenant primitive, such as OP_CAT isn't really a particular behavior. A covenant, such as a vault, is a mere template of behavior built with OP_CAT. And a covenant instance, is a covenant with an exact selection of parameters and data, such as keys, timeouts, and more. Thus keys must be tied to the encumbrance instance, with all preset parameters defined, and not a more general program such as "vault", unless the template parameters are retrieved during evaluation from e.g. the state caboose.

2.1.2. Signing Transactions

The oracle signs transactions only if they comply with the covenant logic.

There is no need to verify a direct relationship between the program-tweaked key of the encumbrance instance and the transaction itself. This is because the transaction input already commits to the key it should verify against. In the worst-case scenario, an attacker might create a signature valid for that key, but such a signature would still be unable to spend the coins if it is not bound to the transaction. If it were bound, the program instance would be valid.

In simpler terms, the oracle only certifies that the transaction complies with the specified program—it does not certify that the transaction requires the provided signature.

```

fn oracle_sign(&self,
               public_key: PublicKey,
               transaction: Transaction,
               witness: Witness) -> Option<Signature> {
    let program_hash = SHA256(self.encumbrance_instance.program);

    if self.encumbrance_instance(transaction, witness) {
        let derived_key = Derive(public_key, program_hash);
        Some(Sign(derived_key, transaction))
    } else {
        None
    }
}
```

Strengths and Limitations **Advantages:**

(I) **Non-Interactive Contract Specification:** Bitcoin scripts for a particular encumbrance can be defined offline without requiring any interaction (after establishing root public key) with the oracle server. This also means covenant keys can be generated from smart contract code without requiring connectivity.

(II) **Stateless operation:** Oracles don't need to track any external database, just decide to sign based purely on local information. The oracle's role is purely mechanical, simplifying implementation and minimizing attack vectors.

(III) **Multi-party flexibility:** Federated multisig configurations can be used to spread out trust assumptions and increase liveness.

(IV) **Presigning Friendly:** For a known state transition, signatures can be generated ahead of time and stored till use.

Limitations:

(I) **Trust assumptions:** There may or may not be oracles worthy of trust.

(II) **Liveness** Except in cases that are enumerable, like CTV, users have to rely on oracles for liveness and may wish to have a longer term failover plan.

OP_CAT and Generalized Conditions For covenants involving concatenation or computational predicates (e.g., OP_CAT or OP_CSFS), the oracle can handle these conditions off-chain. This can be achieved by defining the encumbrance instance as the corresponding Bitcoin script and treating the witness data as the transaction's witness. The oracle then validates the transaction against the script, and if it passes, signs the transaction. The script interpreter used can have whichever opcodes are desired.

For example, the below snippet demonstrates how this might be accomplished.

```
fn oracle_sign_for_cat(&self,
                       public_key: PublicKey,
                       transaction: mut Transaction,
                       witness: Witness) -> Option<Signature> {
    // Derive the tweaked key based on the program hash
    let program_hash = SHA256(self.encumbrance_instance.program);
    let tweaked_key = Derive(public_key, program_hash);

    // Update the transaction input with the encumbrance program and witness
    transaction.inputs[0].witness_program = self.encumbrance_instance.program;
    transaction.inputs[0].witness = witness;

    // Validate the transaction against the provided script flags
    if validate_tx(transaction, SCRIPT_VERIFY_CAT | defaultflags) {
        Some(Sign(tweaked_key, transaction))
    } else {
        None
    }
}
```

State-Based Covenants Stateful covenants, such as roll-ups, may involve the oracle maintaining external state to reference during execution. However, such designs introduce the risk of creating a system that functions as an opaque "black box", reducing transparency for users. An alternative approach leverages cryptographic commitments embedded within the transaction itself to manage state transitions.

This methodology, referred to as the *state caboose*, operates as follows: the final output of a transaction includes an OP_RETURN data carrier, which commits to a hash or a compact data vector representing the program's state variables or database. The funds are encumbered under a fixed program P. When P is spent, its covenant logic retrieves the caboose data from its prior state and subsequently updates or re-commits to a new caboose reflecting the modified state. The primary output of the transaction continues to commit to the same program P, ensuring state integrity and continuity.

The following example illustrates the logic required for a state caboose to manage a unilateral exit in a shared UTXO context, encompassing both state retrieval and update mechanisms.

```rust
fn caboose_read_write(&self, tx: &mut Transaction, input_index: usize) {
    // Retrieve the parent transaction from the witness data
    let parent_tx = tx.inputs[input_index].witness[0];

    // Ensure the input references the parent transaction correctly
    assert_eq!(tx.inputs[input_index].txid, parent_tx.txid());
    let caboose_commit: CabooseCommit =
        parent_tx.outputs
                 .last()
                 .unwrap()
                 .scriptPubKey;

    // Extract the caboose from the witness data
    let caboose: Caboose = tx.inputs[input_index].witness[1];

    // Verify that the caboose matches the committed state
    assert_eq!(caboose.commit(), caboose_commit);

    // Extract and unpack the state row from the caboose
    let row: (Amount, (Key, Amount)) = caboose.get_row();
    let (balance, (user, user_balance)) = row;

    // Retrieve additional witness data:
    // signature, amount, and recipient address
    let signature: Signature = tx.inputs[input_index].witness[2];
    let amount: Amount = tx.inputs[input_index].witness[3];
    let recipient: Address = tx.inputs[input_index].witness[4];

    // Ensure the amount to be sent does not exceed the user's balance
    assert!(amount <= user_balance);

    // Update the transaction outputs
    tx.outputs[0].scriptPubKey = tx.inputs[input_index].scriptPubKey;
    tx.outputs[0].amount = balance - amount;

    tx.outputs[1].scriptPubKey = recipient.to_scriptpubkey();
    tx.outputs[1].amount = amount;

    // Write the updated state row into a new caboose
    let updated_row = (balance - amount, (user, user_balance - amount));
    let new_caboose = caboose.write_row(updated_row);

    // Commit the new caboose state to the final transaction output
    tx.outputs[2].scriptPubKey = new_caboose.commit();
}
```

Zero-Knowledge Encumbering As the complexity of covenant computation increases, particularly when verifying constraints for multiple users, the cost of enforcing such conditions on-chain can become prohibitively high.

To address this, the encumbrance program can be represented as a zero-knowledge verifier. In this approach, the witness data required to satisfy the covenant is replaced by a zero-knowledge proof that is computationally equivalent to the native program. Employing zero-knowledge proofs (ZKPs) effectively reduces the computational burden on the oracle when handling more sophisticated covenants.

Additionally, this method enhances privacy by obfuscating sensitive details. For instance, in roll-up applications, the use of ZKPs allows balances and other sensitive data to remain undisclosed while still ensuring the validity of the covenant's conditions.

Careful consideration must be given to ensuring that when zero-knowledge (ZK) arguments are employed to facilitate state transitions within a contract, all relevant parties retain access to the information necessary to propagate subsequent state transitions.

This requirement is particularly critical in scenarios where state transitions depend on cryptographic commitments, as these commitments may obscure underlying data while preserving verifiability. If any party lacks the requisite inputs or derived proofs to generate the next valid state, the contract could become unusable, leading to deadlocks or loss of funds.

Composable Covenants Covenants can be composed within the creation of the encumbrance instance arbitrarily.

However, in situations where it is desired to isolate different checks to different oracles, covenants can be composed with boolean monotone logic (i.e., AND and OR only). [3]

Multiparty Security and Liveness Composability extends to joining *identical* checks from multiple oracles for the purpose of minimizing risk of a single oracle failure. The semantics of AND under such a federation are such that if any single oracles is honest, the covenant is enforced. This is very useful in contexts where users of the covenant are available to produce signatures as well, as the trust can be minimized. However, with more signers under AND, the risk of a liveness fault increases as if a signature is refused to be provided it may not be.

It may be desirable to compose the covenants such that after a long period of contested non-liveness, the coins are distributed to stakeholders or otherwise saved. Such solutions can be quite sophisticated, the discussion of which is outside the scope of this paper.

[3]It is possible that using MuSig or another multi-party computation scheme, multiple oracles keys might be condensed into a single key, which they sign interactively with to produce a single signature. Using MuSig in this way is an optimization that needlessly complicates the exposition of the concepts in this paper, but would be worth developing for real-world deployments.

These extensions highlight the flexibility of the oracle-assisted covenant framework. While not as secure as native protocol changes, it provides a practical and experimental pathway for deploying advanced covenant functionality today. Further, these covenants are quite efficient as arbitrary rules can be enforced with a single on-chain signature.

2.1.3. Emulating CTV with Oracle-Assisted Covenants

To provide a concrete illustration of the generic framework described above, we will examine its application to CheckTemplateVerify (CTV). CTV is a proposed Bitcoin opcode that enforces transaction templates, making it a suitable example for demonstrating how the framework handles covenant enforcement. Unlike other applications that require additional witness data to validate state transitions, CTV uses the transaction itself as the witness, ensuring that the template's structure inherently validates compliance. By applying the oracle-assisted model to CTV, we can showcase the practical steps involved in replicating its functionality. Similar functionality was implemented in **Sapio**[Rub21] to emulate CTV.

Key Tweak and Emulation Workflow Oracle-assisted covenants replicate CTV functionality by enforcing transaction templates through derived keys and off-chain logic. The workflow proceeds as follows:

(I) **Oracle Initialization:** The oracle generates a root public key $\mathcal{K}_{ctv}$ and commits to using it exclusively under predefined covenant rules.

(II) **CTV Emulation with Templates:** A user provides the oracle with a transaction T and a computed template hash H, derived deterministically (e.g., $H = DefaultCheckTemplateVerif$ to define the exact transaction structure to enforce. There is no additional witness data, V, required.

(III) **Key Tweak for Template Enforcement:** The oracle derives a tweaked key C_T from the root public key K using H:

$$C_T = Derive(\mathcal{K}_{ctv}, H).$$

This derived key replaces the native CTV clause in the transaction script.

(IV) **Transaction Signing:** To spend the output, the user submits the unsigned transaction to the oracle. The oracle recomputes the hash H, derives the private key c_T (corresponding to C_T) and signs.

This method achieves a tight functionality mapping to CTV. By relying on deterministic hashing and key tweaks, the oracle enforces spending conditions structurally, minimizing the need for intelligent server logic.

Public Key Derivation via BIP-32 and Hash Functions In this approach, tweak derivation leverages BIP-32 hierarchical deterministic (HD) keys and a structured hash-to-path conversion process. On initialization, a server generates a seed k and derives a root public key $\mathcal{K}$, which it then publishes. Users pre-plan a transaction T and compute a CheckTemplateVerify (CTV) hash $H = DefaultCheckTemplateVerifyHash(T)$ representing the specific template logic to enforce. To derive a unique key corresponding to H, the hash is converted into a BIP-32 derivation path D consisting of eight 32-bit unsigned integers ($u32$) and one 8-bit integer ($u8$). See Appendix A for details of the conversion process. Although other tweaking mechanisms work, BIP-32 is preferred because it simplifies compatibility with standard signing devices. Importantly, as the derivation is unhardened, it can be computed without contacting the Oracle server.

2.2. Economic Incentive: Fraud-Proof with BitVM

As outlined in the preceding sections, the Covenant Oracle is not inherently penalized for signing transactions that deviate from its intended programming or specified encumbrances.

To enforce honest behavior and accountability, we propose a BitVM-style mechanism that enables operators to detect and prove instances of oracle misconduct, effectively disincentivizing fraudulent activity.

However, a modification to the oracle's design is necessary to ensure that the witness data used in the transaction is accessible to the BitVM operator. Without this adjustment, a malicious fraud proof could exploit the system by falsely claiming an alternative witness or by asserting the validity of a signature without providing the associated witness data. Potential alternatives and mitigations for this limitation are discussed further in Section B.

The Covenant Oracle first declares their key $\mathcal{K}$ and type of covenant they are able to process, $\mathcal{E}$.

$\mathcal{E}$ is restricted such that their must always be a caboose output, of which the Oracle reserves the last data push for arbitrary use.

Then they have the following functions:

```
fn get_instance(self,
                e:Encumberance,
                args: Args)
-> Program {
    e.compile_with(args)
}
```

The oracle signing function is modified to expect the caboose to contain the witness, and to sign the witness data in the caboose.

```
fn oracle_sign(&self,
               public_key: PublicKey,
               transaction: Transaction)
-> Option<Signature> {
    // Prepare the transaction and extract the witness from the caboose
    let (transaction_for_verification, witness) = {
        let mut cloned_tx = transaction.clone();
        let mut caboose = cloned_tx.outputs.last()?.as_caboose();
        let witness = Witness::deserialize(caboose.pushes.pop()?);
        (cloned_tx, witness)
    };

    // Verify the encumbrance instance and sign the transaction if valid
    if self.encumbrance_instance(transaction_for_verification, witness) {
        let program_hash = SHA256(self.encumbrance_instance.program);
        let tweaked_key = Derive(public_key, program_hash);
        Some(Sign(tweaked_key, transaction))
    } else {
        None
    }
}
```

Now, a function is defined which will detect if the transaction was signed by the oracle when the covenant did not pass.

```
fn oracle_malfunctioned(self,
                        k:PublicKey,
                        mut tx:Transaction,
                        s: Signature,
                        e:Encumberance,
                        a: Args
) -> bool {
    let e_i = self.get_instance(e, a);

    let v: Witness = tx.out[-1].as_caboose().pushes[-1].deserialize();
    tx.out[-1].as_caboose().pushes.pop();
    if e_i(tx, v) {
        // no malfunction, there should be a sig
        return false
    }
    // if signed, malfunction
    return verify_sig(Derive(K, SHA256(e_i.program)), tx, s);
}
```

Subsequently, the function **oracle_malfunctioned** should be deterministically transformed into a circuit representation, denoted as $\mathcal{G}$, which can be expressed using NAND gates. This representation enables its integration into a BitVM framework.

The Covenant Oracle should engage a BitVM Operator to create a BitVM instance, utilizing pre-signed transactions, which incentivizes the operator by offering payment if they can produce a valid witness proving that the Covenant Oracle has fraudulently signed a transaction, as determined by the circuit $\mathcal{G}$. Conversely, if the Covenant Oracle can successfully refute the operator's claim by disproving the execution of $\mathcal{G}$, the Oracle is entitled to redeem its locked funds.

In addition, the BitVM Operator is required to post a bond to ensure honest behavior. If the operator equivocates—by revealing more than one bit during execution or providing an incorrect bit—they risk forfeiting their bond, with the associated funds being burned as a penalty. This mechanism enforces integrity and accountability within the BitVM dispute resolution process.

To redeem the bond, a pre-signed transaction exiting the protocol must be broadcast on-chain. This action initiates a delay clock of duration τ, enforced through built-in lock times. The delay provides sufficient time for any evidence of misbehavior, such as fraudulent behavior or equivocation, to be published and adjudicated before the bond can be claimed.

Users of the system are empowered to ensure accountability by submitting evidence of misconduct to the Integrity Oracle if the Covenant Oracle ever cheats. In such scenarios, users gather the necessary proof, which typically includes the fraudulent transaction, its associated signature, and any witness data required to demonstrate a breach of the covenant logic. This evidence is then submitted to the Integrity Oracle, which acts as an impartial adjudicator.

The Integrity Oracle validates the claim by re-executing the covenant logic against the submitted evidence. If the claim is substantiated, the Covenant Oracle is penalized via the slashing of the bond. This mechanism ensures that dishonest behavior carries significant financial consequences, improving the integrity of the system and deterring potential fraud.

3. Discussion

The proposed mechanism provides a robust economic framework designed to enforce the integrity of the Covenant Oracle and deter dishonest behavior. At the outset, the oracle locks funds into a Bitcoin bond, thereby committing to adhere to the specified covenant logic. This bond serves as both a guarantee of good behavior and a source of financial accountability. If the oracle signs a transaction that violates the covenant's conditions, any participant can present proof of the violation, such as the fraudulent transaction and its signature, to initiate a dispute process against the bond.

Upon verification of valid proof, the funds locked in the bond are subject to punitive measures. Depending on the implementation, the funds may be burned, distributed as fees, or allocated to a designated fraud-proving entity or restitution fund, thereby disincentivizing fraudulent activity. To ensure fairness, the oracle retains a pre-signed transaction, allowing it to redeem the bond in two distinct phases, provided no valid evidence of misbehavior is presented within the designated dispute period.

Additionally, the mechanism incorporates safeguards against dishonesty by the BitVM operator. The operator is required to post a stake, which is forfeited if they act dishonestly, such as by equivocating or providing false proofs during the resolution process. This ensures mutual accountability between the oracle and the operator, creating a balanced incentive structure.

This mechanism provides a robust and transparent framework for enforcing compliance with covenant logic. By integrating financial penalties and well-defined verification processes, it ensures that all parties have aligned incentives, fostering both the integrity and reliability of the oracle's operations.

3.1. Advantages of the Oracle-Assisted Model

(I) **Practical Implementation Today**: This model works with existing cryptographic and blockchain tools.
(II) **No Soft Fork Required**: All computation occurs off-chain, avoiding the need for consensus-layer changes.
(III) **Economic Accountability**: The fraud-proof mechanism ensures trust in the system, even if the oracle is untrusted.
(IV) **Optimistic**: In the happy path, none of the fraud proof logic is ever executed

3.2. Drawbacks and Limitations

(I) **Trust in Oracles**: Although mitigated by the fraud-proof mechanism, the model still requires reliance on an oracle's infrastructure.
(II) **Complexity**: The system introduces additional moving parts, such as the oracle and the fraud-proof process.
(III) **Data Availability**: The requirement to post the witnesses on-chain means no space savings compared to bespoke opcodes for covenants

3.3. Future Directions

(I) Continued research into practical FE implementations to replace the oracle model.
(II) Enhancing the fraud-proof mechanism for better scalability and usability.
(III) Exploring other applications of this hybrid model, such as privacy-preserving covenants.

4. Conclusion

While native Functional Encryption remains impractical, the proposed oracle-assisted model enables covenants today. By combining off-chain computation, and an economic incentive mechanism, we achieve a flexible and enforceable covenant system that adheres to Bitcoin's core principles.

5. Acknowledgments

With thanks to Ethan Heilman, who inspired this concept by suggestion that fraud proof checking might be applied to ColliderScript[Hei24].

References

[Hei24] E. Heilman. *Slashing Covenants*. Bitcoin-Dev Google Group Post. Available at: https://groups.google.com/g/bitcoindev/c/nrgqIXL2Cyk/m/9epQ8CMcAwAJ. 2024.

[HKLP24] E. Heilman, V. I. Kolobov, A. M. Levy, and A. Poelstra. *ColliderScript*. Online. Available at: https://colliderscript.co/colliderscript.pdf. 2024.

[Kom24] M. Komarov. *Bitcoin PIPEs: Covenants and ZKPs on Bitcoin Without Soft Fork*. [[alloc] init]. Available at: https://www.allocin.it/uploads/placeholder-bitcoin.pdf. 2024.

[Lin23] R. Linus. *BitVM*. Online. Available at: https://bitvm.org/bitvm.pdf. 2023.

[Rub21] J. Rubin. *BIP-119 Emulator*. Sapio Lang Documentation. Available at: https://learn.sapio-lang.org/ch05-01-ctv-emulator.html. 2021.

[Rub24] J. Rubin. *FE'd Up Covenants*. Online. Available at: https://rubin.io/bitcoin/2024/05/29/fed-up-covenants/. 2024.

Appendices

A. Hash to Vec

In order to create a key per program, we need to tweak the root key by the program. While it'd be possible to apply the tweaks in many ways at lower cost, e.g. using the Taproot tweak formula, we opt for a BIP-32 compatible key derivation to maximize compatibility with various signing devices. This section describes the algorithm to convert from program hash to derivation path.

The derivation format consists of 9 **u32** values, where the top bit of each value is set to 0 (for unhardened derivation). The process begins by splitting the hash (big-endian) into eight **u32** segments and masking off the top bit of each. These masked values form the primary path segments.

To maintain all entropy, the top bits of the eight **u32** values are extracted and combined into a ninth **u32**. This ensures the derivation path retains information lost from masking while adhering to the **ChildNumber** format, which uses the top bit to indicate hardened or unhardened keys. This structured approach is necessary to avoid conflicts with the **u31** encoding of **ChildNumber**.

```


fn hash_to_child_vec(h: Sha256) -> Vec<ChildNumber> {
    let a: [u8; 32] = h.into_inner();
```

```rust
    let b: [[u8; 4]; 8] = unsafe { std::mem::transmute(a) };
    let mut c: Vec<ChildNumber> = b
        .iter()
        // Note: We mask off the top bit.
        // This removes 8 bits of entropy from the hash,
        // but we add it back in later.
        .map(|x| (u32::from_be_bytes(*x) << 1) >> 1)
        .map(ChildNumber::from)
        .collect();
    // Add a unique 9th path for the MSB's
    c.push(
        b.iter()
            .enumerate()
            .map(|(i, x)| (u32::from_be_bytes(*x) >> 31) << i)
            .sum::<u32>()
            .into(),
    );
    c
}
```

B. Data Availability (DA)

It is expensive to force the data to be included in the caboose, as it uses valuable base block space.

Instead of utilizing the caboose, the witness could also be committed via a field such as the Annex. This is preferable because the Annex receives the witness discount. However, the Annex is not presently defined and recommended against for use.

As an alternative, were there a trusted DA layer, then the caboose could include only an attestation from the DA layer, and the **oracle_malfunctioned** program would just require that the witness also be provided to the BitVM proof.

References

[Hei24] E. Heilman. *Slashing Covenants*. Bitcoin-Dev Google Group Post. Available at: https://groups.google.com/g/bitcoindev/c/nrgqIXL2Cyk/m/9epQ8CMcAwAJ. 2024.

[HKLP24] E. Heilman, V. I. Kolobov, A. M. Levy, and A. Poelstra. *ColliderScript*. Online. Available at: https://colliderscript.co/colliderscript.pdf. 2024.

[Kom24] M. Komarov. *Bitcoin PIPEs: Covenants and ZKPs on Bitcoin Without Soft Fork*. [[alloc] init]. Available at: https://www.allocin.it/uploads/placeholder-bitcoin.pdf. 2024.

[Lin23] R. Linus. *BitVM*. Online. Available at: https://bitvm.org/bitvm.pdf. 2023.

[Rub21] J. Rubin. *BIP-119 Emulator*. Sapio Lang Documentation. Available at: https://learn.sapio-lang.org/ch05-01-ctv-emulator.html. 2021.

[Rub24] J. Rubin. *FE'd Up Covenants*. Online. Available at: https://rubin.io/bitcoin/2024/05/29/fed-up-covenants/. 2024.