

# ブロックチェーン公開講座 第19回

## ゼロ知識証明その3（応用事例）

---

芝野恭平

東京大学大学院工学系研究科技術経営戦略学専攻

ブロックチェーンイノベーション寄付講座

特任研究員

[shibano@tmi.t.u-tokyo.ac.jp](mailto:shibano@tmi.t.u-tokyo.ac.jp)



# Agenda

---

- ZKが、ブロックチェーンへ応用されている事例を3つ+1個紹介
- インフラ
  - Zcash（入力の秘匿化）
  - ZKEvm（検証コストの簡略化）
- Dapps
  - Email Wallet (ZKEmail)
  - Private IPFSのデータ存在証明, 加工
- ※ 講義で取り扱う情報は実際のプロジェクトのものと異なり、一部不十分な場合があります.
- また、特定の暗号資産の購入を推奨するものでもございません.

# ゼロ知識証明とブロックチェーン

- ゼロ知識証明は、ブロックチェーンの欠点を補ってくれる

## ブロックチェーン

### 透明性

・ブロックチェーンに記録されたTxは全世界に公開されてしまう。

### 計算能力の限界

・スマートコントラクトで計算は可能であるが、限定的である。

## ゼロ知識証明

### 入力の秘匿化

・入力の一部を秘匿化できる。これにより、プライバシーに関わる情報を隠しつつその情報を活用できる。

### 検証コストの簡略化

・どんな計算を回路で行っても、そのプールの検証コストは一定。

※ 任意の情報を隠蔽化しつつ保存できるわけではない。あくまで回路への入力値の一部が隠せるのみ。

※ オフチェーンでの証明生成のコストが大きい。時間もかかるし、大きな回路を実行するにはそれなりのスペックのPCが必要。

# Zcash



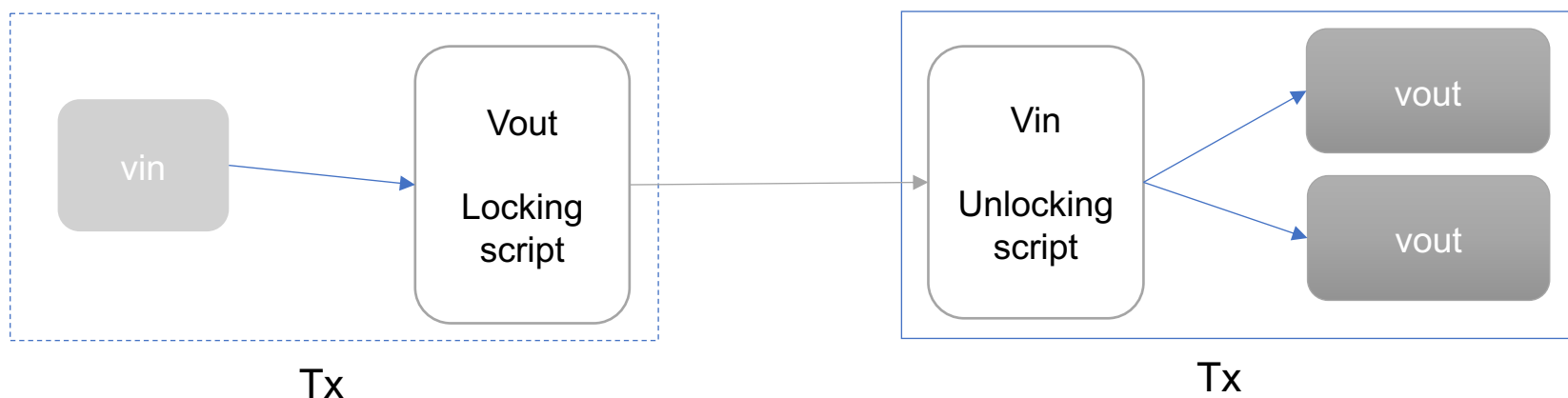
- <https://z.cash/>
- Zcashは、2016年にリリースされたプライバシー重視の暗号通貨である。
- トランザクションの送金元アドレス、送金先アドレス、送金金額が秘匿化される。
- ゼロ知識証明が採用されている。
- 公開トランザクションとシールドトランザクションのいずれかを選択できる。
- UTXOモデルである。
- 本資料は、シールドトランザクションの仕組みについて、sproutという初期のバージョンをベースに作成。

# Zcash

- ビットコインは匿名なのでそれで十分では？
  - ビットコインは匿名であり，個人は自由にアドレスを生成できる．
  - しかしながら，送金履歴はすべてブロックチェーンに記録されているため，特定の個人・法人にアドレスの紐付きがわかると，それまで過去に行った送金の履歴がわかってしまう．
    - 直接店舗などでビットコインで支払いをしたときには，お店側からすると「客である自分」とその使用アドレスの紐付きはわかってしまう．
- 2種類のプライバシー
  - Confidential
    - 送金金額が秘匿化されている．
  - Anonymous
    - 送金者，送金先が秘匿化されている．
- Zcashにおいては，ConfidentialとAnonymousの両方を実現している．

## Zcash : 復習, ビットコインのUTXOモデル

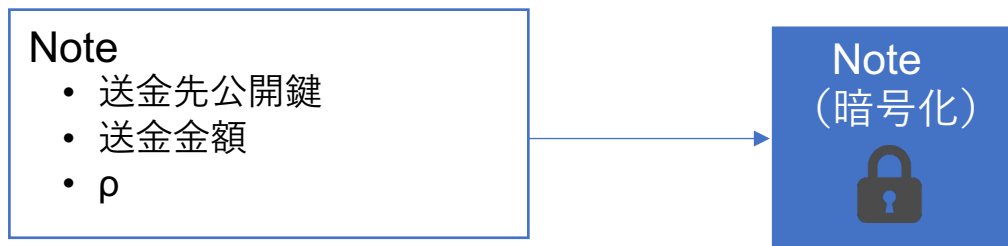
- Zcashはビットコインと同じくUTXOモデルを採用している.



- 送金者は相手の公開鍵ハッシュを元にロッキングスクリプトを作成し, アウトプットとして作成
  - 公開鍵ハッシュはアドレス中に埋め込まれているため, 送金先アドレスを知っていればロッキングスクリプトの作成ができる.
- 送金を行う際には対応するアウトプットに含まれるロッキングスクリプトに対するアンロッキングスクリプトをインプットとして付与.
  - アンロッキングスクリプトは, 秘密鍵による署名を含む.
- Tx内のインプットの和とアウトプットの和が一致することで, 送金金額がおかしくないことを保証している.

# Zcash : 全体像

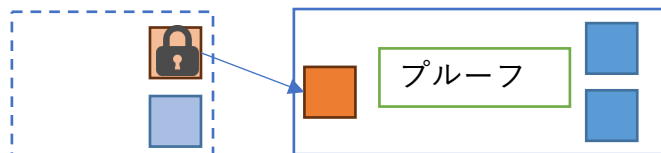
## 送金金額・送金者の秘匿化



- Confidential実現のために送金情報を暗号化.
- トランザクションOutputとしてはNoteの暗号化データを記録.

※ 送金Outputに含まれる情報はNoteと呼ばれます

## 送金時の正当性チェック



- 暗号化されたNoteをInputとしてOutputを作成.
- Outputを作成している本人は自身の秘密鍵でNoteの復号化はでき、矛盾のないOutputを生成できるが、それを第三者に中身を公開せずに矛盾のないOutputを生成していることを示す必要がある.
- これを、ZKを用いて実現. つまり、TxにはZKプルーフが含まれる.
  - Tx内で一つのプルーフが含まれている.
  - Txの正当性に関わる部分をZKで実現している.
- ZKだけだと2重支払いの防止が検出できないため、Nullifierという仕組みを用いて実現

# Zcash : 秘匿化のための暗号化と鍵管理

送金者



送金先



① シールドアドレスを送信

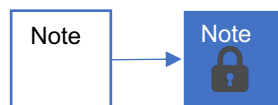
シールドアドレス中には公開鍵が含まれる。シールドアドレスは送金の度に新規生成。(使い回すとプライバシーの問題が生じる)

② ephemeralKeyを生成

送金先が復号化をするためには送金者の公開鍵が必要。送金時にインタラクティブに共有をするのではなく、1度のみしか使わない鍵を生成し使用。公開鍵はTxに含まれる。

③ Note暗号化

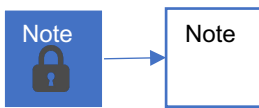
送金者はDH鍵共有で生成された共通鍵で、Noteを暗号化。



④ 暗号化Noteと ephemeralKey (公開鍵)はTx内に

⑤ 送金先は自身の秘密鍵で復号化

送金先は自身の秘密鍵と、Tx内の ephemeralKey (公開鍵)を用いて共通鍵を生成、それを使ってNoteを復号化。AEAD\_CHACHA20\_POLY1305



※補足) DH(Diffie-Hellman)鍵共有

相手の公開鍵と自身の秘密鍵を用いて共通鍵を生成する仕組み。

Alice



Bob



① Bobの公開鍵を送信

② Aliceの公開鍵を送信

Aliceの秘密鍵とBobの公開鍵で共通鍵を生成

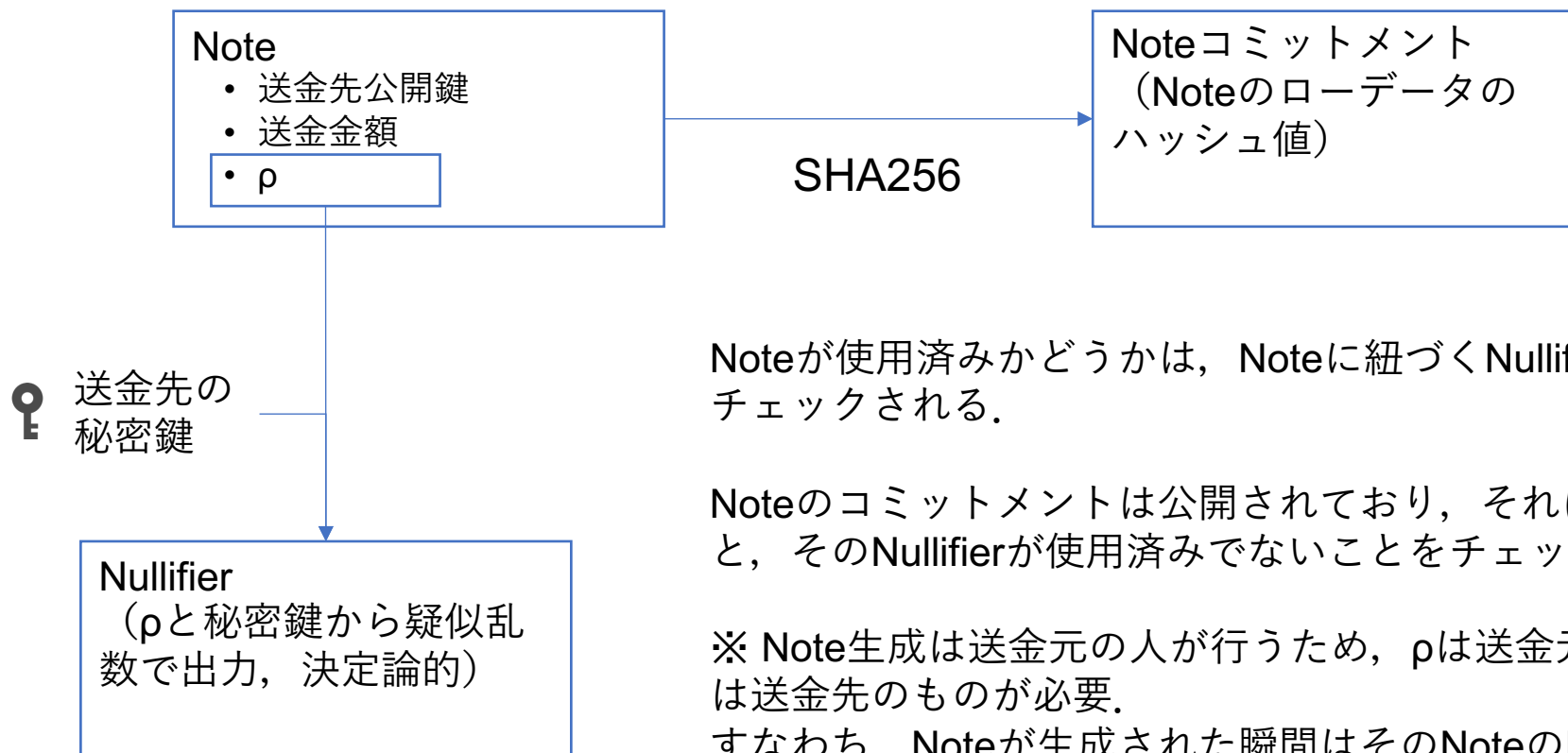
Bobの秘密鍵とAliceの公開鍵で共通鍵を生成

同じ共通鍵を生成できる

※ Zcashでは送金用のキーペアと、暗号化用のキーペアの2種類が使用されています。本講義では、簡略化のため一つのキーペアとして話を進めます。



## Zcash : Noteと送金済みフラグの分離 (Nullifier)



Noteが使用済みかどうかは、Noteに紐づくNullifierがすでに公開されているかどうかでチェックされる。

Noteのコミットメントは公開されており、それに紐づくNullifierを送金者が知っていること、そのNullifierが使用済みでないことをチェックし、2重支払いを防止している。

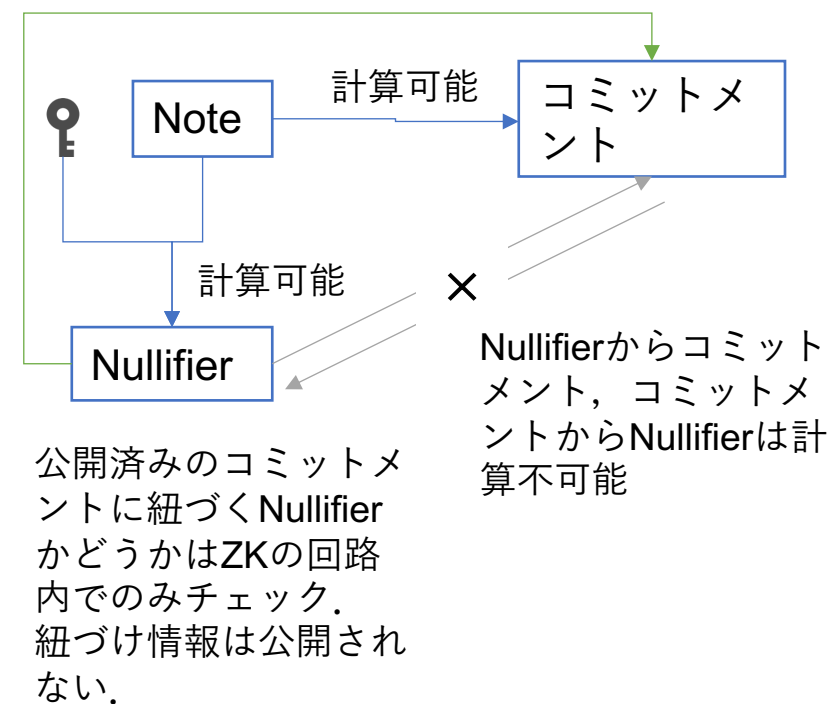
※ Note生成は送金元の人が行うため、 $\rho$ は送金元が決める。Nullifier計算のための秘密鍵は送金先のものが必要。

すなわち、Noteが生成された瞬間はそのNoteのNullifierは誰も知らない。

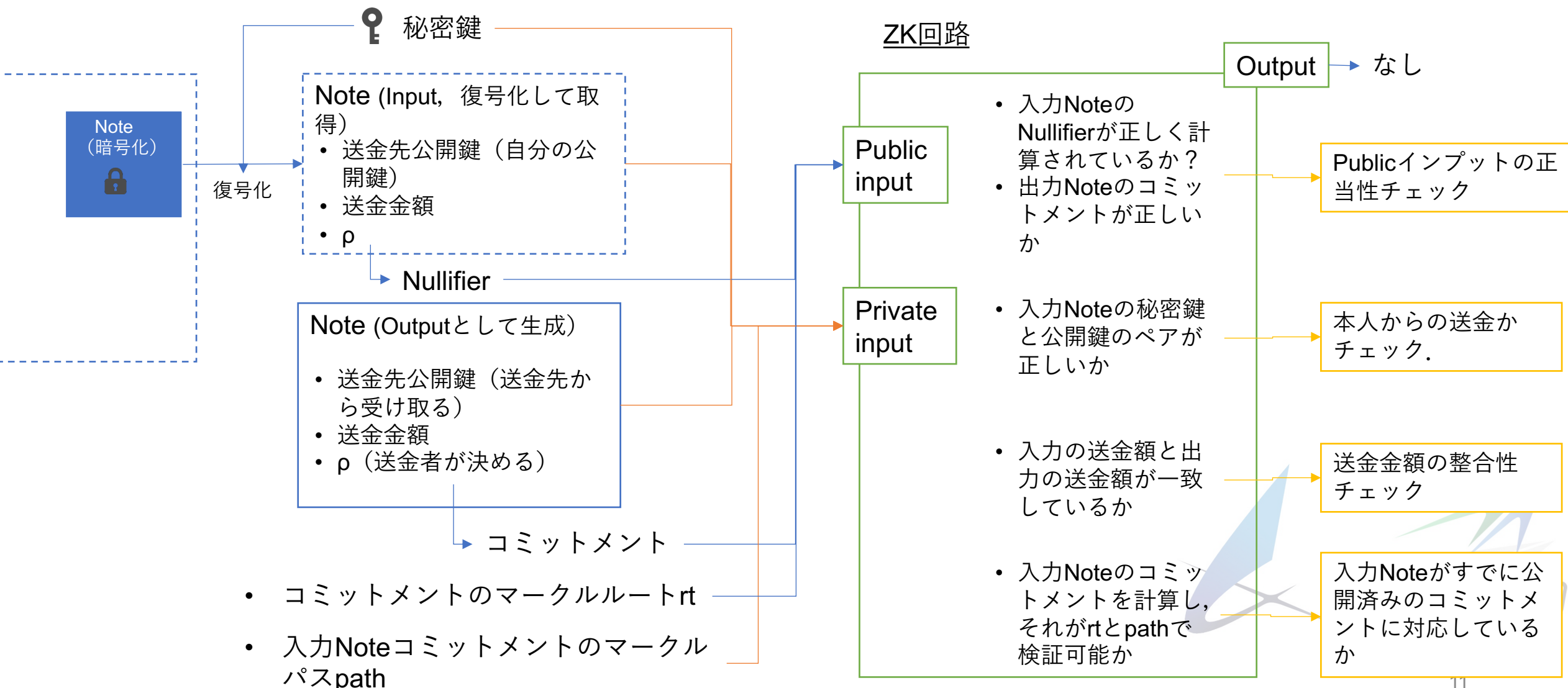
Nullifierは一意的な値に決定される、それとコミットメントの紐づけはZK回路内の計算により保証される。

## Zcash : 2重支払いの防止のためのNullifierとNoteコミットメントツリー

- Noteの代表値としてコミットメントが公開されており、これはマークルツリーで管理されている。
  - ZK回路内でコミットメントの有無を判定しているためマークルツリーを使用
- コミットメントには紐づくNullifierが存在しており、使用済みNullifierを含むTxは無効とみなされる。
  - Nullifierは特にマークルツリーでは管理されていないが、過去のブロック中のTxをチェックしNullifierが存在しないことでチェックできる。
- つまり、コミットメントの値で送金済みかどうかのチェックをしているわけではなく、それと完全に推論が不可能なNullifierという識別子で送金済みかどうかのチェックをしている。
- コミットメントは公開されているが、使用済みコミットメントはわからない。
- 仮に使用済みフラグをNullifierではなくコミットメントで行うようにした場合、使用済みNote、未使用Noteがわかるようになってしまう。
  - 超小規模で運用しているケースにおいても、消去法的に誰のTxかどうかは類推しにくくなっている。



# Zcash : Noteに対して送金の正当性を示すためのZKプルーフ

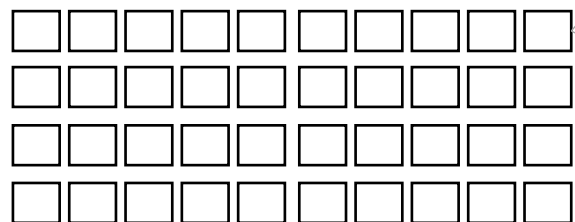


## Zcash : ブロックとトランザクションの構造

- ブロックチェーンに記録される情報から、第三者は送信者や送金に関する情報は何も得られない。

### ブロック

- 前ブロックハッシュ
- PoWのナンス
- PoWターゲット
- Txルート
- Noteコミットメントルート



### トランザクション

#### ZK

- インプットNoteに関するNullifier (複数)
- アウトプットNoteのコミットメント (複数)
- 送金が正しいというプルーフ
- 消費したNullifierに対するコミットメントが存在しているNoteコミットメントルート
- 暗号化されたアウトプットNote (複数)  
上記のコミットメントと同じ個数.
- EphemeralKey (公開鍵)  
Note暗号化のための送金者の公開鍵
- 署名  
署名の鍵はTx毎に乱数で生成.  
改ざん耐性をつけるため付与.  
ビットコインのように公開アドレスはないため, 「誰が」つけた署名なのかは問題にならない.

## Zcashまとめ

---

- 送金内容を暗号化
  - Confidential（送金金額の秘匿化）， Anonymous（送金者の秘匿化）の実現
- 暗号化されたデータ内の処理の正当性
  - ZKを使って実現.
  - ZK回路内で，送金者の正当性と送金金額の正当性のチェックをしている.
- 2重支払いの防止
  - Noteごとにユニークな値である，コミットメントとNullifierの2つを持って匿名化を保ちつつ実現.
- 参考URL:
  - <https://z.cash/learn/who-created-zcash/>
  - <https://www.binance.com/ja/research/projects/zcash>
  - <https://cir.nii.ac.jp/crid/1050292572094389888>
  - <https://zips.z.cash/protocol/sprout.pdf>



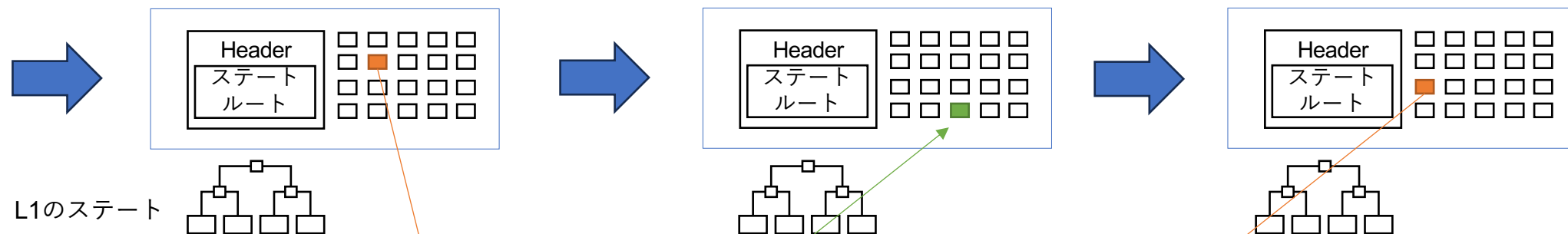
## zkRollup/zkEvm

---

- 今後は、ZKの特長のもう一つである「検証コストの簡略化」を活用した事例.
- Ethereumのスケーリングソリューション.
- Optimistic Rollupが計算結果の検証を行っていなかったのに対して、zkRollup/zkEVMでは、計算前の状態から、計算後の状態に移ったことがZKにより保証される.
- zkEVMは、様々なプロジェクトが様々なアプローチにより実装を行っている.
- 本講義では、基本的な考え方をベースに紹介.

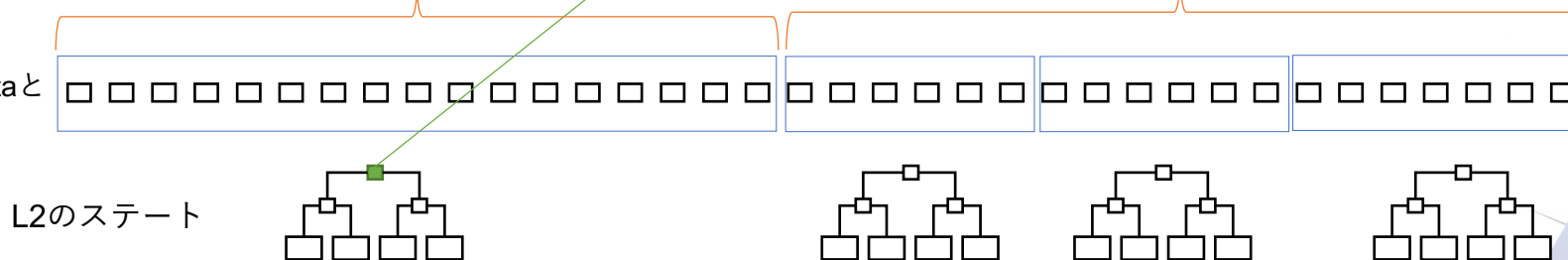
# zkRollup/zkEvm：Rollupでの処理イメージ図（再掲）

## L1ブロックチェーン



## L2

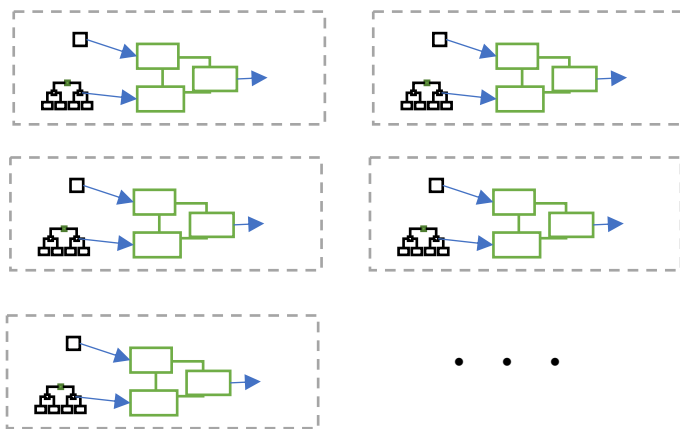
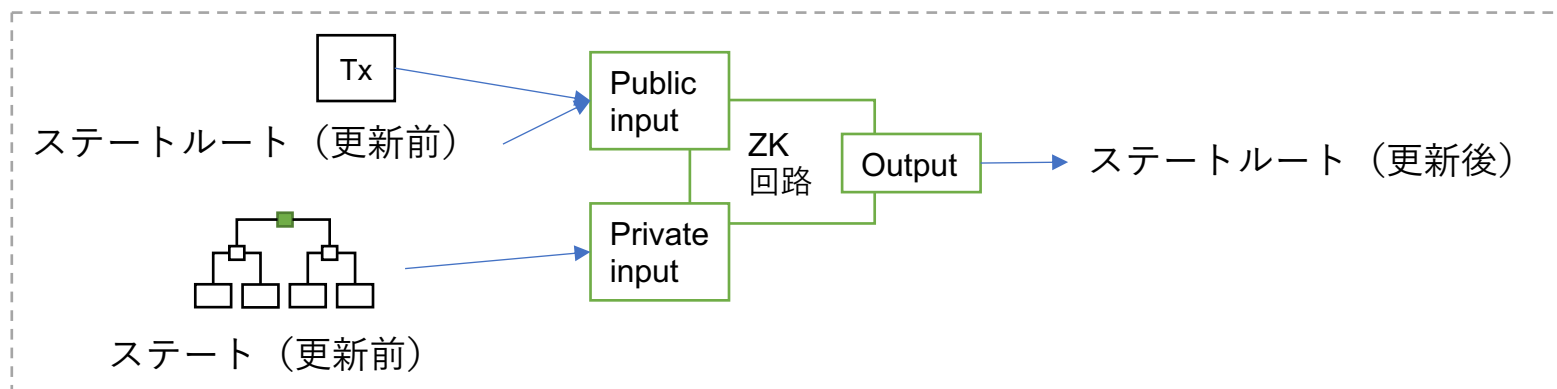
L2TxはL1にcalldataとして書き込まれる。



L2ステートは、L2ブロック内のL2Txを実行して更新。  
何らかのタイミングでL1にステートルートが記録。

# zkRollup/zkEVM：シンプルな実装アイデア

- 必要な処理をZKの回路で記述し、その実行とステートの更新を行うことで任意の計算に対して検証コストを一定に抑えることが実現可能



## 問題点

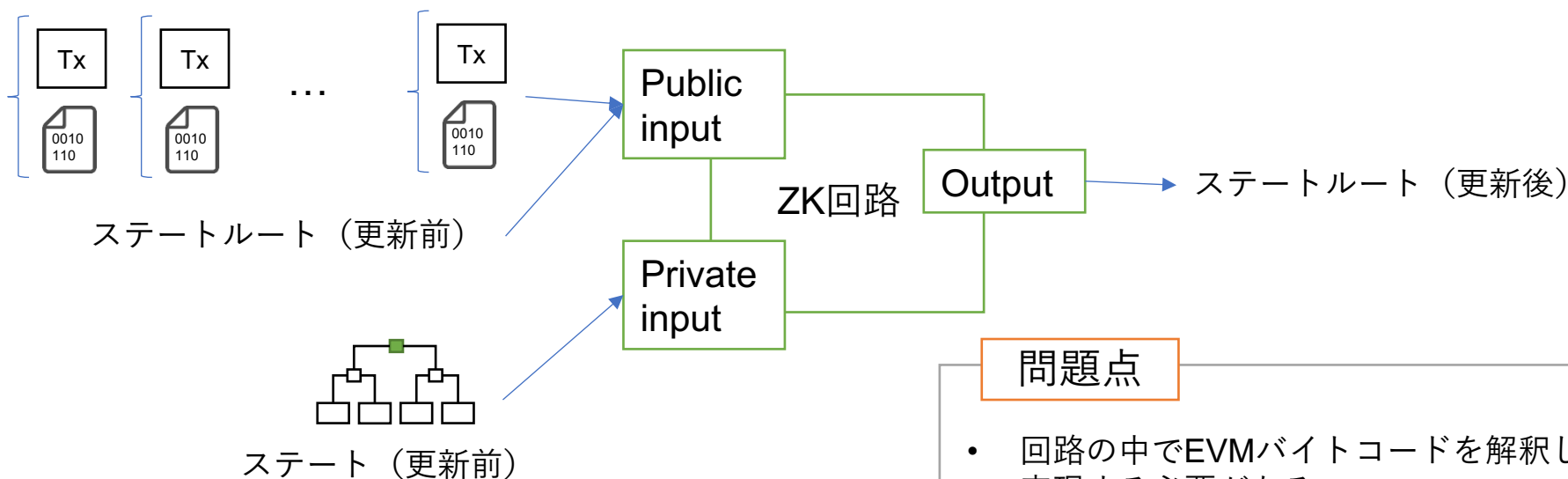
- 実行したい関数分だけZKの回路を作る必要がある。
  - すでにデプロイされていて運用されているコントラクトなどが流用できず、すべて作り直しになってしまう。
  - ソースコードの流用もできず、circomその他専用コードを作り直す必要がある。
- 1回ずつの実行結果をコントラクトで検証することになるので、そんなに処理能力向上に繋がらない。



## zkRollup/zkEVM：シンプルな実装アイデア2（ベースアイデア）

- Txとそれに関連するコントラクトのバイトコードをすべて入力し，更新後のステートルートを出力する．
- パブリックインプット・アウトプットとプルーフをL1コントラクトに記録．

トランザクションと，コントラクトのEVMバイトコードのペア



- Txとそれを実行するコントラクトのバイトコードをすべてパブリックインプットに入れて，変更後のステートルートを出力する
- 検証者は，バイトコードがL1にデプロイされているコントラクトのバイトコードと一致しているかどうかをもとに実行処理が正しいかを確認可能

### 問題点

- 回路の中でEVMバイトコードを解釈した上での実行処理を実現する必要がある．
- 非常に巨大な回路になる上に，入力データも多くなり，プルーフ生成の計算処理が問題になる．
- コントラクトのバイトコードを常にパブリックインプットに入れると，L1のコントラクトでの検証時にデータサイズが問題になる．

## zkRollup/zkEVM : EVMバイトコードの回路内の処理

- まずはEVMで計算処理がされる処理を理解する. EVMバイトコードは, Opcodeによる状態遷移で表現される.
- 2つの値の和を取るadd()という関数が入っているコントラクトについて考える.

```
contract AAA {  
    function add(uint a, uint b) public pure returns (uint256) {  
        return a + b;  
    }  
}
```

これをコンパイルして, バイトコードに変換したものが以下:

0x608060405234801561001057600080fd5b5060405160208061...

### 6080 - PUSH1 0x80

これはスタックに値0x80をプッシュするオペコードです。スタック操作の一部で、メモリ管理やエラー処理のために使われる。

### 6040 - PUSH1 0x40

スタックに値0x40をプッシュします。メモリ領域の操作に関連する。

### 52 - MSTORE

PUSH1 0x40で指定されたメモリ位置に0x80をストアします。これにより、メモリの一部が予約されます。

### 34 - CALLVALUE

トランザクションで送信されたEtherの量を取得します。一般的に、関数がパブリックでEtherを受け取れる場合に使われる。

Program Counter	Opcode	Stack	Memory	Storage
53	PUSH1 0x80	[ , , 80]	...	...
54	PUSH1 0x40	[ , 40, 80]	...	...
55	MSTORE	[ , , ]	...	...
56	CALLVALUE	[ , , value]	...	...
57	DUP1	[ , value, value]	...	...
58	PUSH2 0x0010	[ 0x0010, value, value]	...	...
...				

## zkRollup/zkEVM：EVMバイトコードの回路内の処理

- zkEVMにおいて、Zkの回路内でやることは以下の2つ。
  - オペコードの処理の証明（EVM証明）
  - スタックやメモリの状態がオペコードによって正しく更新されたことの証明（状態証明）


回路へのInputとしてEVMバイトコードと状態遷移表の2つが入力され，その1行1行の結果がOpcodeをEVM証明することによって正しく推移していることを証明する．

Program Counter	Opcode	Stack	Memory	Storage
53	PUSH1 0x80	[ , , 80]	...	...
54	PUSH1 0x40	[ , 40, 80]	...	...
55	MSTORE	[ , , ]	...	...
...				

EVMバイトコードから得られる

状態遷移表はInputから与える.

EVM証明  
EVM証明

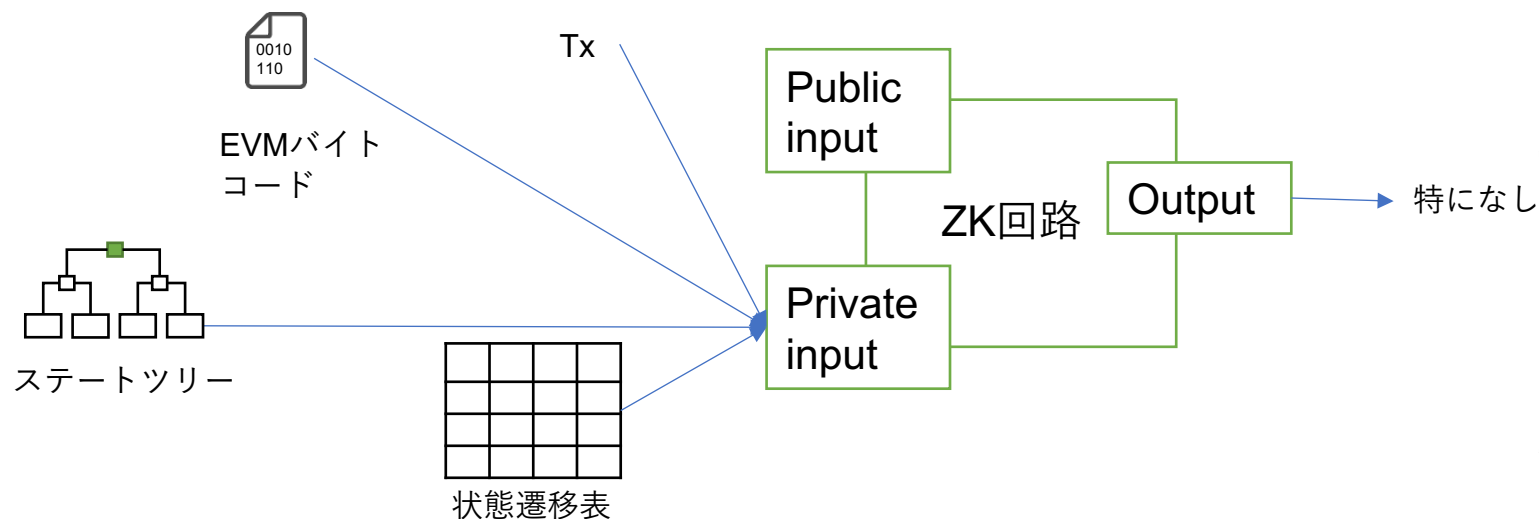




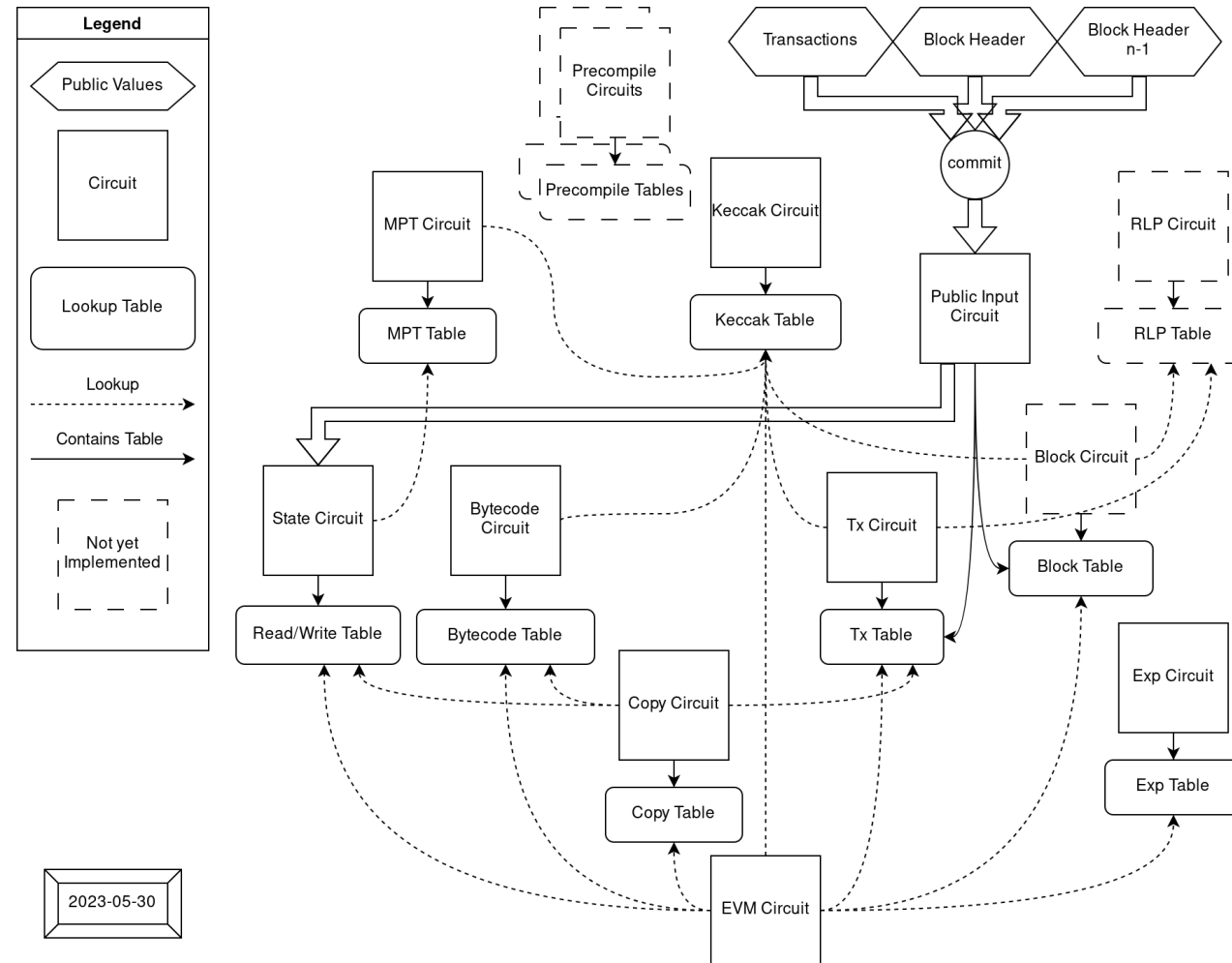
## zkRollup/zkEVM : EVMバイトコードの回路内の処理

- 必要な回路への入力情報
  - EVMバイトコード
  - 状態遷移表
  - 関数の入出力値
    - ステートツリーの内部で各コントラクトのストレージエリアの情報管理がされており、変数のストレージの位置が特定できる.
    - 入力値は, Txから抽出できる.

※ 状態遷移表をインプットとして入力するためには, 一度回路外でEVMバイトコードを実行してすべての状態遷移表を取得する必要がある.



## zkRollup/zkEVM：回路全体像



<https://privacy-scaling-explorations.github.io/zkevm-docs/architecture.html>

## zkRollup/zkEVM：プルーフ生成時のパフォーマンス向上技術

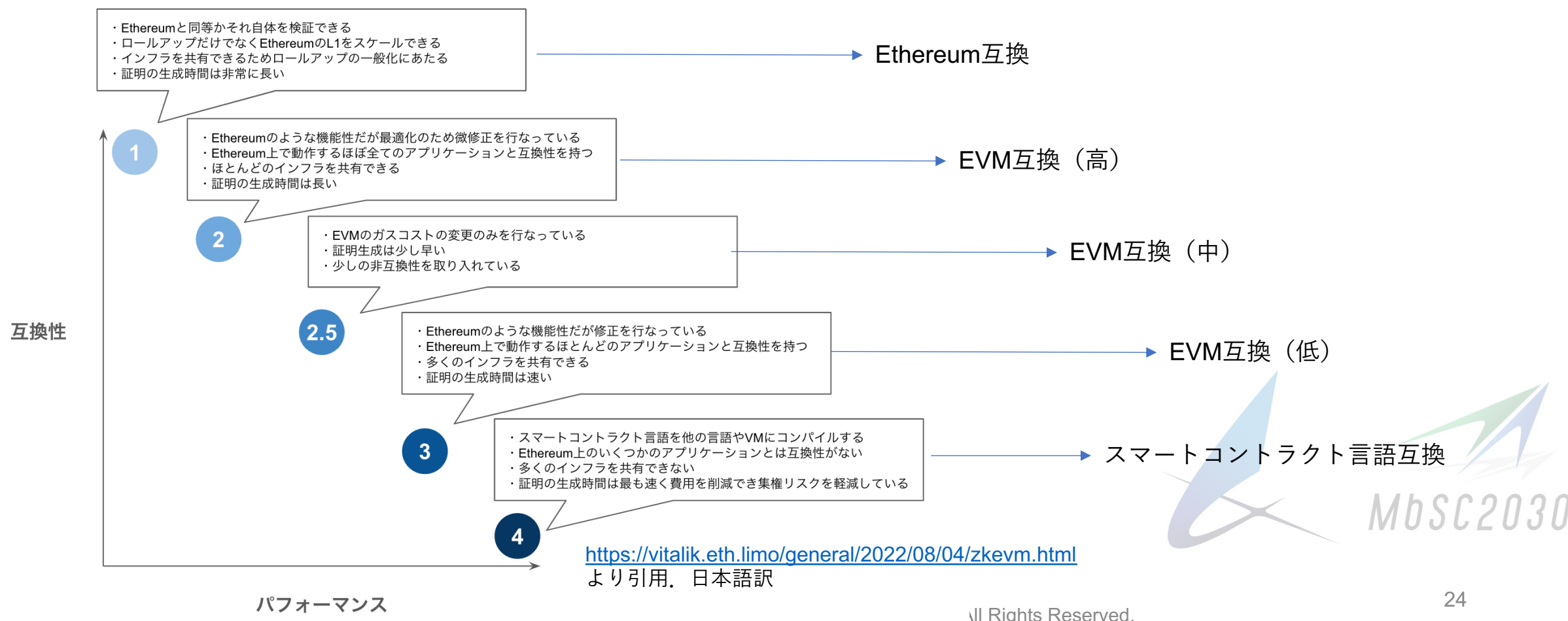
- Opcodeによっては回路内での処理が大きく時間がかかる。（zk unfriendlyな処理）
  - ハッシュ関数Keccak256などビット演算が行われる計算はオーバーヘッドが大きい。
  - また、マークルパトリシアツリーの操作もデータの位置の特定やハッシュ計算が含まれるためオーバーヘッドが大きい。
- これらのzk unfriendlyな処理の処理速度を上げるには：
  - プルーフ生成の新しいアルゴリズムの構築や、GPU、ASICを利用した計算高速化
  - 特定のオペコードについて、回路内で処理することを諦める
    - 完全EVM互換ではなく、似たような機能で代替する，ということ。
    - ZKフレンドリーなハッシュ関数の利用など。
  - Lookupテーブルの利用

## zkRollup/zkEVM：プルーフ生成時のパフォーマンス向上技術

- リカージョン
  - 回路内でProof検証を行う
  - 例えば, Tx一個ずつのProofを生成し, それを複数個まとめ上げて1つのProofに集約する.
  - 1個ずつのProof生成について, 並列処理が可能なものもある.
- フォールディング
  - 連続したいくつかの処理をまとめて1つの制約にできる.
  - 例えば, Txに対して, 1つずつ制約を生成し, プルーフ生成まで処理を行ってしまうのではなく, 複数個まとめて一つの「制約」を生成する.
  - その制約に対して, 多項式変換, コミットメント, と処理を行う.
  - リカージョンよりもさらに計算コストの低減が期待できる.
- Lookupテーブル
  - 回路内で計算を行うのではなく, あらかじめ計算した結果の一覧表をInputとして与えて回路内では検索をして利用する.
  - 例) SHA256の計算は時間がかかるから, あらかじめある値の範囲のSHA256を計算し, Inputとして与える
    - 0から $2^{256} - 1$ までのSHA256を計算し, 一覧表にする, 等.

## zkRollup/zkEVM：処理速度向上のための戦略

- すべてのEthereumの機能をZKで実現するとプルーフ生成コストが高い。
- zkEVMプロジェクトによってその対処の方法が異なり， VitalikはzkEVMを5つに類型している。





## zkRollup/zkEVM：まとめ

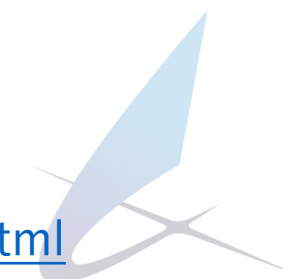
---

- Optimistic Rollupと同じように複数のTxをいくつかまとめてステートの更新を行う。
- その際に、ZKにより計算の正しさを保証し、検証コストを抑えることを実現している。
- Ethereumの機能をそのまま回路内で実現することは、計算量的に困難。
- 様々なプロジェクトがそれぞれのアプローチで解決を試みている。

## zkRollup/zkEVM : 参考URL

---

- <https://cypherpunks-core.github.io/ethereumbook/13evm.html>
- <https://www.evm.codes/playground>
- <https://scroll.io/blog/zkevm>
- <https://vitalik.eth.limo/general/2022/08/04/zkevm.html>
- <https://immutableblog.medium.com/ground-up-guide-zkevm-evm-compatibility-rollups-787b6e88108e>
- <https://medium.com/@blockchain101/solidity-bytecode-and-opcode-basics-672e9b1a88c2>
- <https://speakerdeck.com/alexcj96/dive-into-scroll-zkevm>
- <https://www.ethervm.io/>
- <https://ethereum.org/ja/developers/docs/evm/opcodes/>
- <https://docs.scroll.io/en/developers/ethereum-and-scroll-differences/>
- <https://www.evm.codes/precompiled>
- <https://privacy-scaling-explorations.github.io/zkevm-docs/architecture.html>



## Email Wallet (ZKEmail) : 概要

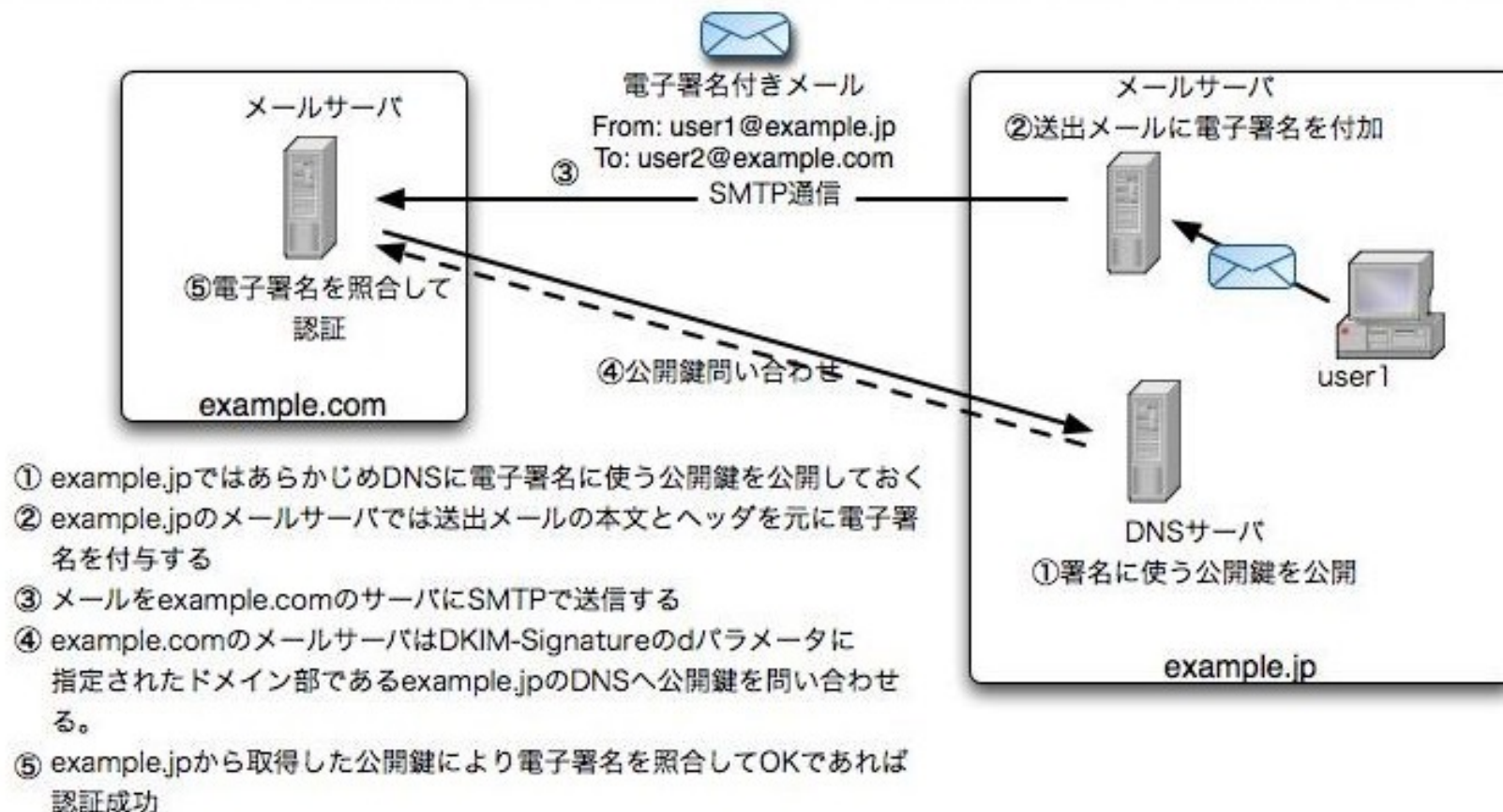
---

- ZKの特長である「検証コストの簡略化」を活用した事例.
- 自分の普段使っているメールアドレスを使って, メールを送信することによりコントラクトウォレットの操作が行えるソリューション.
- 自身のメールアドレスから, 特定の形式のメールを送信すると, 送金やコントラクトの実行ができる.
  - メッセージ形式は正規表現を用いて柔軟な表現が可能.
- Suegami, Sora, and Kyohei Shibano. "Contract Wallet Using Emails." 2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC). IEEE, 2023.
- <https://github.com/zkemail/email-wallet>

# Email Wallet (ZKEmail) : 概要

## 基本的なアイデア

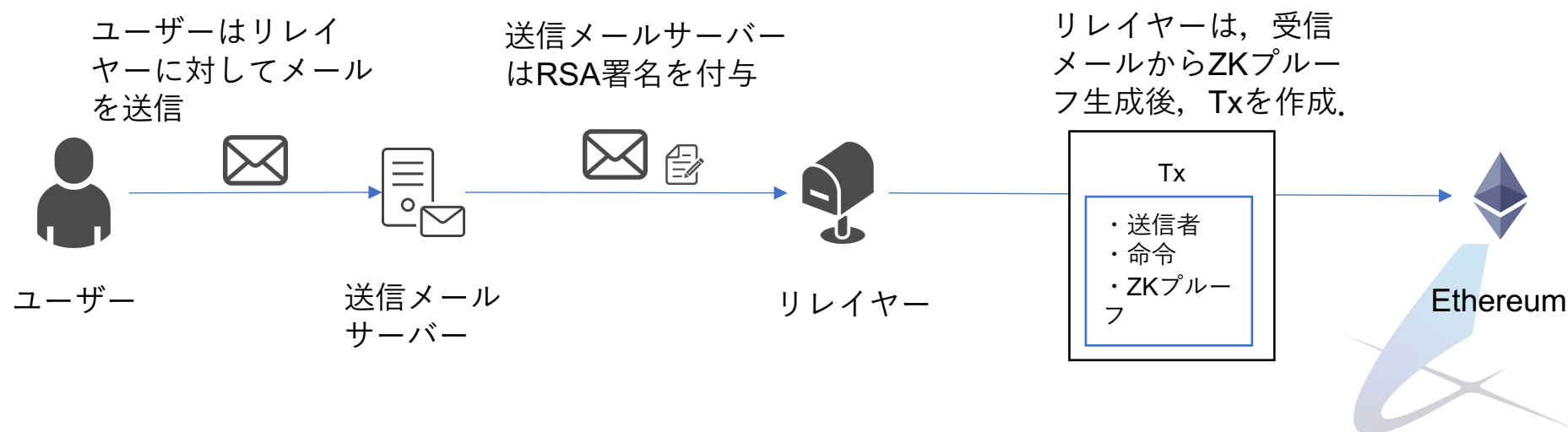
- 既存メールサービスでは、送信メールの改ざんを防止するため電子署名が付与されている。(DKIM)
- メールサーバーがRSA形式の署名を付与.
- この署名を検証することでコントラクト上で資産の移動を可能にできるのでは？
  - メールサーバーのトラストは必要.



[https://salt.iajapan.org/wpmu/anti\\_spam/admin/tech/explanation/dkim/](https://salt.iajapan.org/wpmu/anti_spam/admin/tech/explanation/dkim/)

## Email Wallet (ZKEmail)：システム構成

- 既存のメールサービスの多く（Gmail, Outlook, iCloud等）はDKIMをサポートしている。
  - つまり、メールに対してRSA署名をつけてくれる。
  - メールの送信者がその文章を書いたことをメールサーバーが保証してくれる。
- メールについて、電子署名のチェックを行いその文章が本人から送られたことを検証できる。
  - メールサーバーが、メールの改ざんを行っていない限りそれは正しい。
- メール送信者をアカウントとして、メール本文に記述されている内容を命令として、コントラクトに送信して処理できるようにしたのがEmail Wallet (ZK Email)



# Email Wallet (ZKEmail) : 送金例

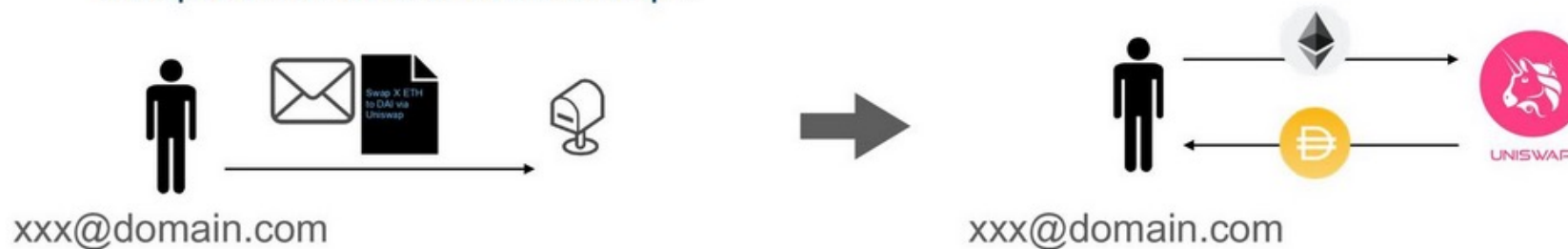
## Example 1

A user can transfer the user's crypto assets deposited to the contract wallet by sending an email with the message "Transfer X ETH to yyy@domain.com".



## Example 2

A user can exchange the user's ETH to DAI via Uniswap by sending an email with the message "Swap X ETH to DAI via Uniswap".



<https://speakerdeck.com/sorasuegami/icbc2023-contract-wallet-using-emails?slide=5>



# Private IPFSのデータ存在証明

- 入力の秘匿化を活用した例.
- 医療情報など、機密情報を含むデータを特定グループ内で管理する方法としてプライベートIPFSがある.
  - 分散型ストレージで、特定のグループ内でデータを管理.
  - データ一つ一つにはcidというユニークなIDが付与されて管理.
- そのデータを外部ユーザーに配布することがあった場合、「本当にそのプライベートIPFSで管理されたデータなのか」を検証可能にするためにZKを利用.
- さらにZK回路内で特定の処理を入れることでデータそのものではなく、加工が可能.
  - 例：
    - 画像データに、配布先の人の名前を埋め込む.
    - 社員の人事情報Excelの、その人の開示可能な情報以外を消されたデータを提供.
- Shibano K, Ito K, Han C, Chu TT, Ozaki W, Mogi G. Secure Processing and Distribution of Data Managed on Private InterPlanetary File System Using Zero-Knowledge Proofs. Electronics. 2024; 13(15):3025. <https://doi.org/10.3390/electronics13153025>

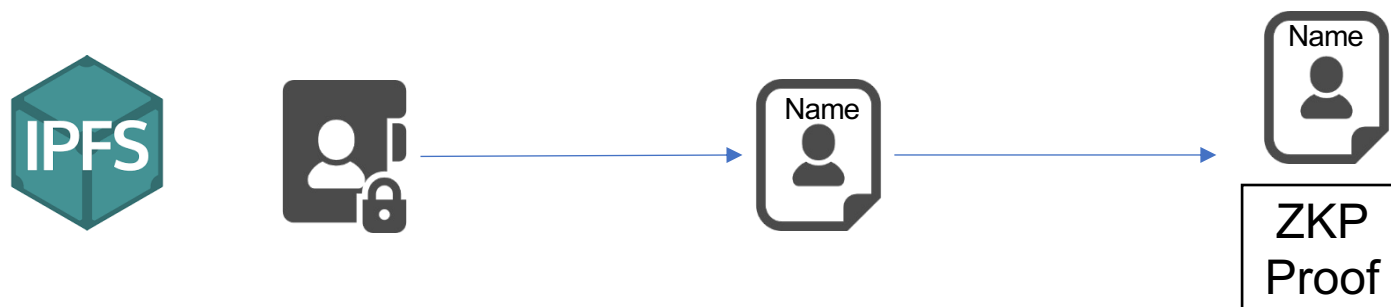
## Private IPFSのデータ存在証明：データ管理の前提

- 画像を暗号化して、プライベートIPFSで保存



※ 共通鍵暗号方式で暗号化，鍵は特定の管理者が管理

ネットワーク外部の人に復号化した画像でかつ，その人の名前を埋め込んだ状態で配布．





# Private IPFSのデータ存在証明：ZK回路での処理



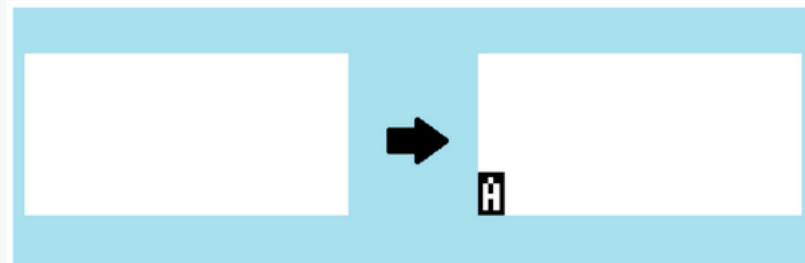
- 外部ユーザーは青い四角で囲まれたところを受け取る。
- Private IPFSにはNWに参加している人しかアクセスできない。
- 外部ユーザーは、受け取ったデータの真正性確認のため、ネットワーク参加者のうち誰か一人でも信用できればよい。
- そのネットワーク参加者にはプライベートIPFSにそのcidが存在しているかどうかをチェックしてもらう。

# Private IPFSのデータ存在証明：計算時間

Table 3. Comparison of the execution time of the circuit.

	Pixel	Image Size [Byte]	Nonlinear Constraints	Build Time [ms]	Proof Gen Time [ms]
Standard					
	10 × 10	384	558,341	668,220	14,377
	15 × 15	784	1,095,292	1,316,666	25,545
	30 × 15	1440	1,980,688	1,696,370	35,161
	30 × 30	2816	3,867,989	3,657,578	65,785
	60 × 30	5456	7,453,300	7,864,583	126,262
ZK-optimized					
	10 × 10	376	35,407	128,979	3076
	15 × 15	775	72,725	173,210	4032
	30 × 15	1435	134,663	275,630	5900
	30 × 30	2815	263,450	470,872	9733
	60 × 30	5455	509,375	850,612	17,571
	60 × 60	10,855	1,013,483	1,257,418	29,044
	120 × 120	43,255	4,036,913	6,754,928	96,580

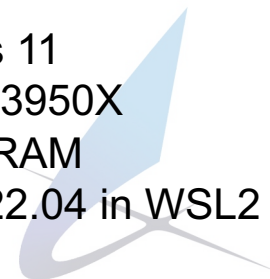
Figure 3. An image generated by the circuit for a 60 × 30 image size by ZK-optimized implementation.



- 120x120ピクセル（ビットマップ）より大きい画像は、RAMが足らずに処理できなかった。
- ZK-Optimizedで120x120ピクセルの画像のビルドに約2時間弱かかった。

## 環境

- Windows 11
- Ryzen 9 3950X
- 128 GB RAM
- Ubuntu 22.04 in WSL2





## まとめ

---

- ZKの応用事例を見た.
  - ブロックチェーンのインフラ系2つの事例（ZcashとZKEVM）
  - Dapps系2つ（zkEmail, Private IPFSへの応用）
- プライバシー保護や，検証コストの簡略化により，ブロックチェーンの応用領域を拡張できていることが確認できた.



- 
- 本スライドの著作権は、東京大学ブロックチェーンイノベーション寄付講座に帰属しています。 自己の学習用途以外の使用、無断転載・改変等は禁止します。