Midnight Matching Pennies

The following explanation details my step-by-step approach to the Matching Pennies exercise with Compact Smart Contract language.

The presentation is divided into three main sections:

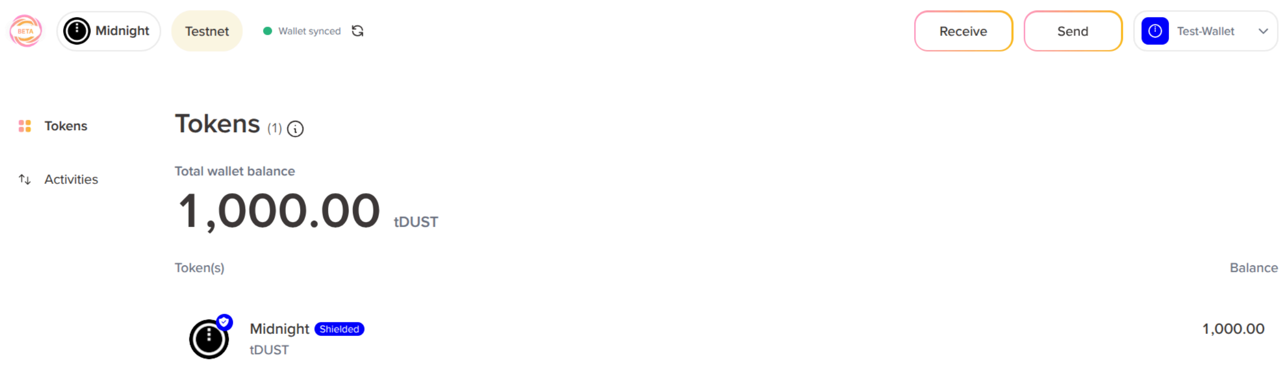
- **Development Environment**
- **Compiling Example Contract**
- **Matching Pennies Implementation**

Section 1: Development Environment Setup

To begin coding our Matching Pennies implementation, we must first set up the correct development environment. Following the Midnight Network Developer Tutorial, the setup process involves:

- Set up Midnight (locally)
    - Lace Wallet.
    - Token acquisition.
    - ZK Proof Server.
- Build a DApp
    - Install Compact Compiler.
    - Building & Running DApp.
- Creating your DApp (In our case, Smart Contract)

Lace Wallet & Token Acquisition
First, we set up our Lace Wallet locally. We then receive a set of Dust currency to perform operations through our DApp network; the Midnight website provides an interface to send free Test Tokens for developers.



Credentials:

- **User:** TestWallet
- **Password:** TestWallet1234

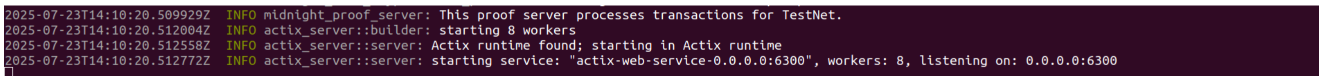
Initializing ZK Server

The *Zero Knowledge* server allows the users to validate, secure and shield transactions inside the network. By simulating a real life network locally, we must initialize the *ZK* server before performing any test operation.

With Docker: After pulling the images from Midnight docker repository, we execute the following command on our terminal.

```
docker run -p 6300:6300 midnightnetwork/proof-server -- 'midnight-proof-
server --network testnet'
```

This ensures we are *correctly* running our proof-server & exposes our container's 6300 default port to communicate over our network and provide an interface to later validate our entire network operations.



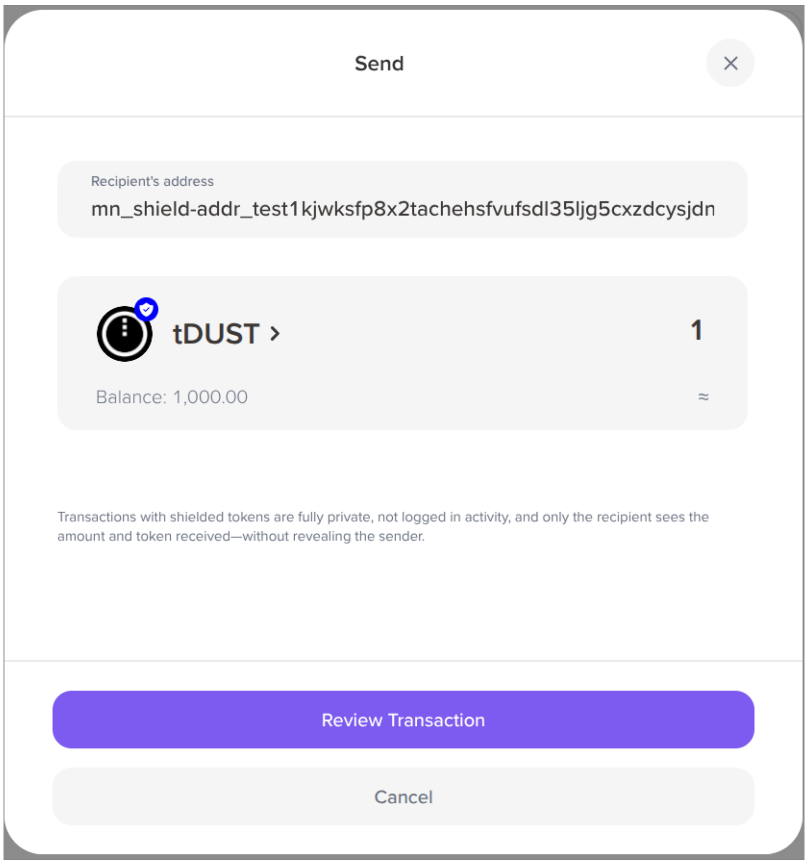
```
2025-07-23T14:10:20.509929Z  INFO midnight_proof_server: This proof server processes transactions for TestNet.
2025-07-23T14:10:20.512004Z  INFO actix_server::builder: starting 8 workers
2025-07-23T14:10:20.512558Z  INFO actix_server::server: Actix runtime found; starting in Actix runtime
2025-07-23T14:10:20.512772Z  INFO actix_server::server: starting service: "actix-web-service-0.0.0.0:6300", workers: 8, listening on: 0.0.0.0:6300
```

Performing our first transaction locally

Following the tutorial, we assume our best friend's wallet is the following address:

```
Address:
addr_test1kjwksfp8x2tachehsfvufsdl35ljg5cxzdcysjdn6ntadspyxn3qxqrxypgjm055
c2azrpuyn7un0ge2vm25vkfv38d24rj3ewcku5wmdc94gjr9
```

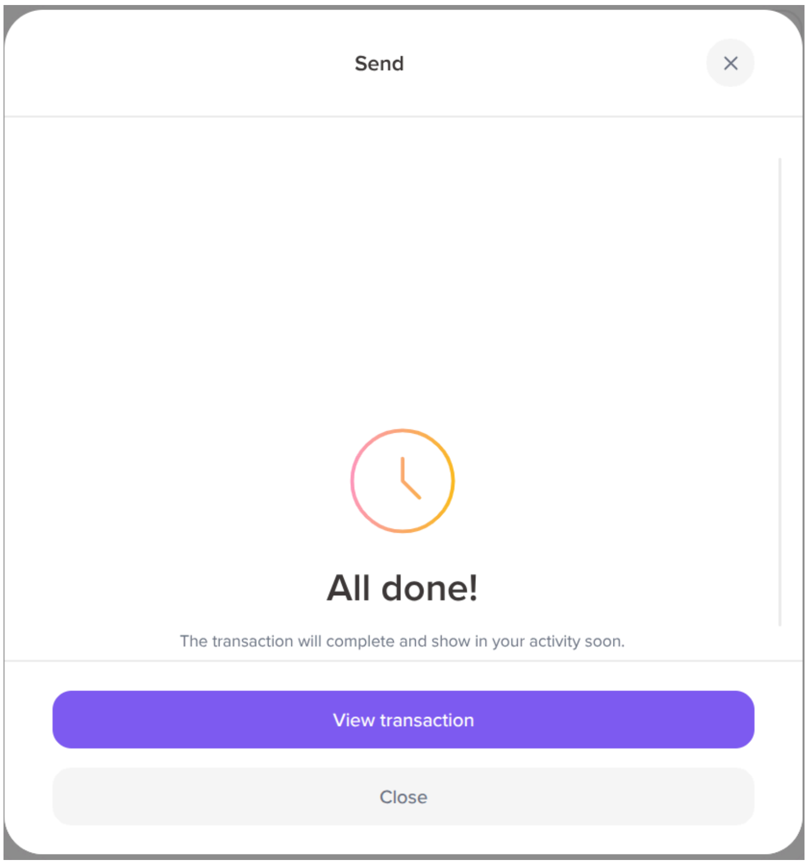
And with that, we've sent our friend 1 tDUST.



mn_shield-addr_test1kjwksfp8x2tachehsfvufsdl35ljg5cxzdcysjdn

Balance: 1,000.00

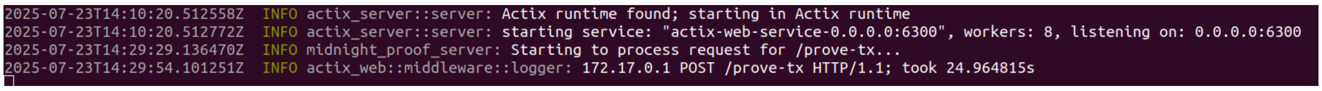
Transactions with shielded tokens are fully private, not logged in activity, and only the recipient sees the amount and token received—without revealing the sender.



All done!

The transaction will complete and show in your activity soon.

We can see that our current ZK server succesfully recieved the request and handled it correctly.

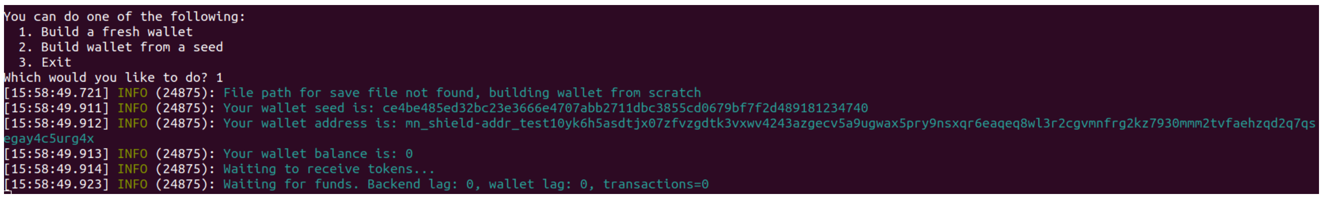


```
2025-07-23T14:10:20.512558Z  INFO actix_server::server: Actix runtime found; starting in Actix runtime
2025-07-23T14:10:20.512772Z  INFO actix_server::server: starting service: "actix-web-service-0.0.0.0:6300", workers: 8, listening on: 0.0.0.0:6300
2025-07-23T14:29:29.136470Z  INFO midnight_proof_server: Starting to process request for /prove-tx...
2025-07-23T14:29:54.101251Z  INFO actix_web::middleware::logger: 172.17.0.1 POST /prove-tx HTTP/1.1; took 24.964815s
```

Section 2: Compiling Example Contract

We want to verify that our local network is correctly handling contracts between two wallets with our example based on the example-counter repository.

After following the documentation step by step, defining environment variables and configuring the Compact compiler path, we proceed with the contract generation process.



```
You can do one of the following:
  1. Build a fresh wallet
  2. Build wallet from a seed
  3. Exit
Which would you like to do? 1
[15:58:49.721] INFO (24875): File path for save file not found, building wallet from scratch
[15:58:49.911] INFO (24875): Your wallet seed is: ce4be485ed32bc23e3666e4707abb2711dbc3855cd0679bf7f2d489181234740
[15:58:49.912] INFO (24875): Your wallet address is: mn_shield-addr_test10yk6h5asdtjx07zfvzgdtk3vxwv4243azgecv5a9ugwax5pry9nsxqr6eaqeq8wl3r2cgvmnfrg2kz7930mmm2tvfaehzqd2q7qs
egay4c5urg4x
[15:58:49.913] INFO (24875): Your wallet balance is: 0
[15:58:49.914] INFO (24875): Waiting to receive tokens...
[15:58:49.923] INFO (24875): Waiting for funds. Backend lag: 0, wallet lag: 0, transactions=0
```

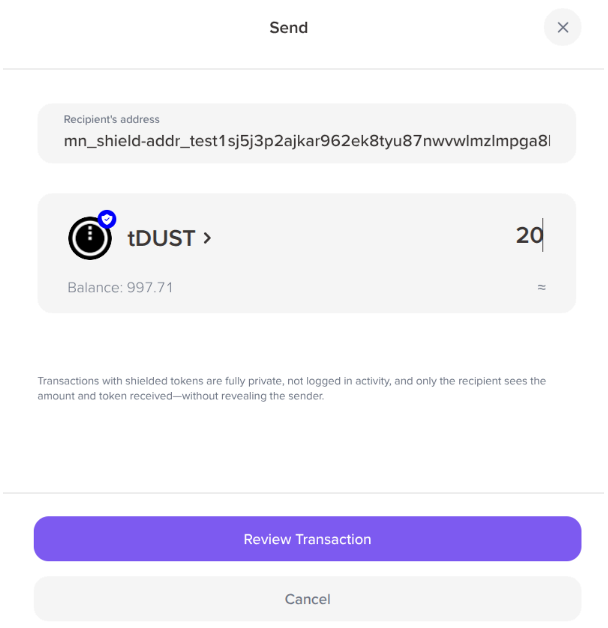
Example generated CLI wallet

```
Example wallet seed:
30c112855c5942f935e821d1d882f4bb4c436876caca1115f8935e77abb17e8b

Example Wallet address:
mn_shield-
addr_test1sj5j3p2ajkar962ek8tyu87nwvwlmzlmpga8l6my7nrf9cn8s4eqxqyp47fjgjwh
c4mse66xgflsumcxq0vznc2wksy4lhkvrzge8nx67v4sg808
```

From our Lace Wallet, we verify that our test network is fully functional by initiating a contract transaction over the blockchain.

1. **Transaction Initiation**: Sending test DUST tokens from Lace Wallet to the generated CLI wallet.
2. **ZK Proof Generation**: The zero-knowledge proof server processes the transaction privately.
3. **Blockchain Settlement**: The transaction is recorded on the Midnight network.
4. **Local Wallet Update**: The CLI wallet receives and confirms the transaction.


Send

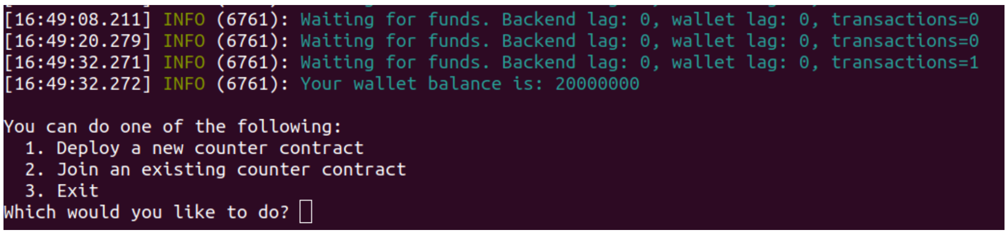
Recipient's address
mn_shield-addr_test1sj5j3p2ajkar962ek8tyu87nwvwlmzlmpga8l

tDUST › 20

Balance: 997.71 ≈

Transactions with shielded tokens are fully private, not logged in activity, and only the recipient sees the amount and token received—without revealing the sender.

And successfully receiving it on our local wallet.


```
[16:49:08.211] INFO (6761): Waiting for funds. Backend lag: 0, wallet lag: 0, transactions=0
[16:49:20.279] INFO (6761): Waiting for funds. Backend lag: 0, wallet lag: 0, transactions=0
[16:49:32.271] INFO (6761): Waiting for funds. Backend lag: 0, wallet lag: 0, transactions=1
[16:49:32.272] INFO (6761): Your wallet balance is: 20000000

You can do one of the following:
  1. Deploy a new counter contract
  2. Join an existing counter contract
  3. Exit
Which would you like to do?
```

This transaction confirms that our connection between the local wallet and Lace wallet has been established correctly, with our ZK server functioning as the middleman to verify the exchange.

Having properly installed our compiler and executed some examples, we are now ready to implement our Matching Pennies game.

Section 3: Matching Pennies

This last section develops my approach to the matching pennies exercise. During the previous sections, we built the foundational understanding of Compact language and Midnight network to later leverage their privacy capabilities on our game.

The entire code is available on the GitHub repository, so if no explanation is desired, the implementation is 100% accessible.

Objective

We want two players (A & B) to pick one side of a coin. In real life, the following task is straightforward, both players pick a side and simultaneously reveal the answer in order to then proceed and toss the coin to decide the winner. No party can interfere with the other player's decision and no player can know each others decision beforehand.

However, based on the nature of blockchains, all ledger information values remain public. Due to this constrain, we want to preserve the privacy of each player's decision before actually revealing their choice.

Structure
In order to play a game of Matching Pennies, two players must join the game. If a third one attempts to connect, the Smart Contract handles the logic to forbid the player's presence.

joinGame & commitCoin
Once two players have joined, our structural approach consisted on setting up a *Commitment Scheme*. Both players have to make a decision on a coin side, to which they later commit and cannot change the decision.

This commitment ensures our players have decided a coin side *a priori* but we have no insight on what the value holds.

- We enforce this commitment policy with a ledger boolean value `player1_commitment & player2_commitment` that verifies whether a player has previously committed or not. No player can commit more than once per match.

How is our commitment safe?

Our implementation leverages Compact in built utilities & functions. More specifically, on our model, we commit a hashed value based on the *coin* side & *salt* value which provides randomness to the hash. Moreover, Compact provides `persistentCommit<T>` function specifically to securely commit values. Docs on commitment.
Having our players safely committed their decision, they will later proceed to reveal their coin side.

revealCoin
Once both players have committed to their coin choices, they proceed to the reveal phase. In this phase, each player submits their chosen coin side (heads or tails) along with the salt used in their commitment. The revealCoin circuit verifies that:

- The lobby is full and no external players can reveal a coin, except for *player1* & *player2*.
- The player hasn't previously revealed his choice.
- Comparing the stored commited value provided by the commit circuit to the *coin* & *salt* produced commitment.

In addition, we can safely input *coin* & *salt* values as a function parameter due to Compact's nature of privacy on passed circuit parameters; therefore, a player can safely call *revealCoin* with their secret values without worrying external observers accessing the data.

playMatchingPennies
Last, we want to determine the winner of our current game.

The *playMatchinPennies* circuit determines the winner of the Matching Pennies match based on the game's rules. It ensures a fair outcome enforcing:

- All players must be connected.

- All players must commit to their answers before revealing.
- All players must reveal their answers before playing.
- No player can reveal before the other commits (this last rule enforces other players not picking a coin side after the other player reveals his choice).

If these conditions are met, the circuit verifies:

```
if(player1_coin_choice == player2_coin_choice){
    winner is player1 // Proper logic inside code
} else {
    winner is player2
}
```

In order to allow another match to take place, we reset the default values of the ledger to accept new games.

Code Compilation
We can verify this code successfully compiles by running the compiler on the *MatchingPennies.compact* file.

Checking the compiler version is easy.

`compactc --version // expected output: Compactc version: 0.24.0`

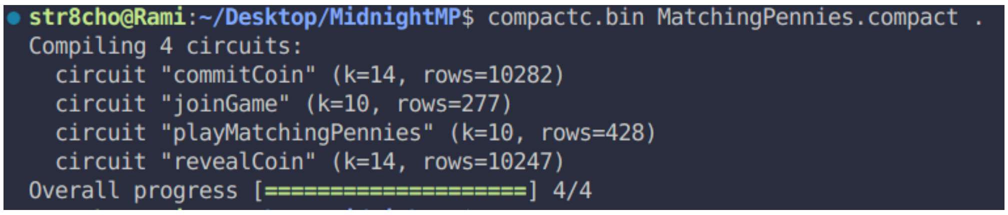
And once our compiler is installed, we run:

```
compactc.bin MatchingPennies.compact .
```

We can expect the following folder structure.

```
MidnightMP
├── compiler
├── contract
├── keys
├── MatchingPennies.compact
├── node_modules
├── package.json
├── package-lock.json
├── scripts
├── test
└── zkir
```

And compiler's output.



```
● str8cho@Rami:~/Desktop/MidnightMP$ compactc.bin MatchingPennies.compact .
  Compiling 4 circuits:
    circuit "commitCoin" (k=14, rows=10282)
    circuit "joinGame" (k=10, rows=277)
    circuit "playMatchingPennies" (k=10, rows=428)
    circuit "revealCoin" (k=14, rows=10247)
  Overall progress [====================] 4/4
```

Test
We provided a test suit to verify the functionality of the Smart Contract, but given Compact's recentness and close source, testing has resulted in a set of difficult challenges to surpass in the short span of time.

A future development area for Midnight & Compact is to implement a *plug-and-play* testing framework locally so developers do not have to upload contracts to the blockchain & use the Testnet, reducing wasteful time fixing error & improving development experience.

Conclusion
This project, built without prior blockchain experience, demonstrates my ability to quickly learn a new language, adapt to its unique features, and deliver a working solution under time constraints.

In addition, it has opened a new set of tools to develop and enrich my tool-case as a software developer and explore unknown regions of programming such as decentralized networks and blockchain technologies.

Future Improvements & Questions
Along the development of the exercise, questions & possible improvement spots arouse for future opportunities.

- Minimizing exposure by keeping players address private.
- Adding a queue of players waiting to play (if two players are in development) and automatically connect.
- Storing history of players with a different structure (in this case, virtual addresses can be used to infer player's patterns and learn based on their decisions).