Increasing the Mina node’s security by fuzzing the transaction application logic

As software systems grow in complexity, they involve increasingly more components, modules, and dependencies that often interact with each other in intricate ways. For blockchain specifically, due to the *state explosion problem* – the continuous growth of the distributed ledger – conventional testing methods such as formal verification or model checking may not be practical.

We want to maximize the security and stability of the Mina network, and for such a complex and dynamic system, we need to use approaches that cover unique algorithm paths.

One of the methods that are suitable for testing in such complex and challenging environments is *fuzzing*. This method of testing involves generating random inputs and feeding them into the system to trigger unexpected behaviors. This can cover a wide range of inputs, including edge cases that may not be considered during regular testing or through manual analysis.

Fuzzing can better simulate real-world conditions, as it often uses random, unexpected, or malformed inputs that an attacker might use. It is a form of stress testing that pushes the software to its limits and simulates extreme conditions. This can be particularly valuable for complex systems such as blockchains that may have unpredictable behavior under heavy loads or when faced with unexpected inputs. This can help to identify issues such as memory leaks, buffer overflows, and other security vulnerabilities that could be exploited by attackers.

Additionally, we can integrate fuzzing into the software development lifecycle as part of continuous testing and improvement processes. This ensures that the node is consistently tested against new and evolving threats.

What are we fuzzing?

We are specifically targeting the *transaction application logic* which is the code that defines the rules and processes that govern how transactions are created, validated, and recorded on the blockchain. We are testing zkApp transactions, which can do multiple *account updates* (changes to the account’s properties, including changes to its balance) in a single transaction, also they support operations to work with custom tokens.

Fuzzing invariant checks

We want to be able to detect bugs in the logic through fuzzing. To achieve that, we need to encode a set of conditions that can't be broken in the code of our Rust implementation of the Mina transaction application logic. We then fuzz both the Rust and original OCaml implementations, and if some of these new condition checks fail during fuzzing, then we have found a bug in the logic.

We are using *invariants* (properties or values that do not change over the course of the transaction's application) to check if the transaction application logic is checking for these permissions correctly.

Here is a more detailed overview of how the check is performed:

- We added 3 extra functions to our Rust implementation of the application logic. These functions are called every time an AccountUpdate is applied:

1. The first function is called at an early stage in the application logic before any checks or updates are done. This call retrieves the following information from the current AccountUpdate:
    - **The AccountId**: this is a combination of the public key (address) and the TokenID. This is what is used to find a specific account (the same public key can contain multiple accounts holding different tokens). The reason why the AccountId is obtained from the AccountUpdate is to be able to locate the target account in the ledger and inspect its state.
    - **The Authorization**: this is set by the client and can be a signature (signed with the account's owner private key), a Proof (generated by a zkApp), or None. This authorization is compared against the set of permissions in the account (obtained from the ledger by locating it with the AccountId we retrieved before). Then, a set of booleans is produced for each permission that describes whether the current AccountUpdate can or cannot modify the specific parts of an account.

    It keeps track of what kind of modifications are allowed to be performed to the target account by comparing the update's authorization with the account permissions. This call also makes a copy of the target account's state at the point **before** any modifications are done by the application logic.

2. The second function is called right before the end of the application logic. This function makes a copy of the target account's state at the point **after** the potential modifications are performed by the application logic.

3. The third function is called when the last AccountUpdate was processed and the application logic is about to commit all changes to the ledger. Here, we will check all the accounts involved in the transaction. Remember that for each account update, we saved copies of the account's state **before** and **after** the update. We compare these **before**

and **after** states against the account permissions and authorization type of the account update: if there is any change in the account's state that doesn't follow the permission rules, then we have found an invariant violation.

Note that what we are checking is just that the application logic follows the permission model. This can't prevent issues caused by incorrect settings of these permissions by the user – for example, when a zkApp is deployed with a set of permissions that is too permissive. An example of that is when the account's owner set all permissions to `None` which would allow everyone to do whatever with that account (spend its tokens for example) without any kind of authorization (no signature or proof required).

The Mina Permissions Model

Currently, the only invariants we are testing correspond to Mina's permission model for accounts.

To understand the permission model, first, let's define the following terms:

Permissions

Permissions are a part of the account state itself (and there are also permissions to update permissions: `setPermissions`). They are required to make further changes to the account and they can be set to: `Proof`, `Signature`, `Either` or `None`.

Authorizations

The *authorization* is set in an account update (a zkApp transaction is composed of one or more account updates). Every operation or update that we want to apply to some part of the state of an account requires a specific authorization:

a) `Signature` - the signing of the transaction (this can only be done with the secret key of the account's owner)
b) `Proof` - signing of the transaction with a proof (generated from the zkApp)
c) `None` - no authorization is needed

The transaction application logic checks the authorization used in each account update (and the possible changes this account update does) against the permissions in the account.

A new account can be created by sending enough funds (including account creation fees) to its address (which is the public key of the account). When a new account is created it has a default set of permissions, these permissions are:

```
edit_state: Signature,
  send: Signature,
```

```
            receive: None,
            set_delegate: Signature,
            set_permissions: Signature,
            set_verification_key: Signature,
            set_zkapp_uri: Signature,
            edit_action_state: Signature,
            set_token_symbol: Signature,
            increment_nonce: Signature,
            set_voting_for: Signature,
            set_timing: Signature,
            access: None,
```

We can see that for most operations (all except for receiving tokens) we need a Signature authorization, which means that the transaction must be signed with the secret key that corresponds to the public key (address) of the account.

Let's say we want to transfer funds from one account to another (this is typically done with legacy transactions). This operation involves accessing two accounts (or three, if we specify a fee payer other than the sender): the sender, and the receiver.

To transfer funds, we need to:

1. Subtract the number of tokens we are sending from the sender's account balance: for this to happen, the authorization we use must be accepted by "send" permission of the account. We know that by default it is "Signature" and since we (the user) are the owners of the sending account we can sign it with our secret key.
2. Add the number of tokens we are sending to the receiver's account balance: for this to happen, the authorization we use must be accepted by the "receive" permission of the receiver's account. This is "None" by default, so anyone can increase the receiver's balance without any kind of authorization (it is common to send funds to accounts we don't know thus we don't own their secret key).
3. Finally, we should increment the nonce of the sender, to prevent *replay attacks*. The permission checked is "increment_nonce" which is set to "Signature", so the same authorization we used for "send" will suffice.

If we wanted to transfer funds with a zkApp transaction (instead of a legacy transaction), it requires us to include two "account updates" in the zkApp transaction:

```
// Sender account update
AccountUpdate {
        body: Body {
          public_key: sender_public_key,
          token_id: TokenId::default(), // 1 for MINA tokens
          balance_change: neg_amount_to_send, // -amount_to_send
          increment_nonce: true,
```

```
          use_full_commitment: true,
          implicit_account_creation_fee: false,
          authorization_kind: AuthorizationKind::Signature,
          ..
        }
        authorization: Control::Signature(signature), // signed with
the secret key corresponding to public_key
    }
// Receiver account update
AccountUpdate {
        body: Body {
          public_key: receiver_public_key,
          token_id: TokenId::default(), // 1 for MINA tokens
          balance_change: pos_amount_to_receive, // +amount_to_send
          authorization_kind: AuthorizationKind::NoneGiven,
          ..
        }
        authorization: Control::NoneGiven,
    }
```

The transaction application logic will process every account update (after processing the fee payer information) and apply the corresponding changes to the accounts (decrease balance and increment nonce for the sender's account, and increase balance for the receiver account).

Fuzzing custom tokens

Finally, there is a permission with special handling which is the `access` permission.

Mina has support for custom tokens, here we can have a *token owner* account that is responsible for authorizing account updates to other accounts, which balance is in that particular token and not in MINA.

To authorize these updates, a transaction sends multiple account updates in a tree-like structure: the parent node is an *account update* bound to the token owner's account, with a list of children account updates for accounts holding that particular token. These children account updates require not only the authorization for the target account but also the authorization of the parent (the token owner account).

How we are fuzzing Mina

Differential fuzzing is used between the OCaml implementation of the transaction application logic and our port in Rust of the same logic. It is meant to find inconsistencies between both implementations.

In Mina, the ledger is represented by a data structure known as a Merkle tree. Inconsistencies between the OCaml and Rust implementations are detected by applying the same transaction to both implementations and then comparing the Merkle root hash of the two ledgers.

The fuzzer runs in an infinite loop of generating and applying random inputs until it finds an issue. If there has been a large number (currently 5000) of iterations without finding an issue, then coverage information is used to detect whether the fuzzer is making further progress or not.

How fuzzing progress is measured

To measure progress, the fuzzer reads the full list of coverage counters (in the Rust from LLVM coverage information, and in OCaml from bisect-ppx):

- if any of the counters whose value was zero in the previous measurement is now not zero, it means we hit a new code path, which a sign of progress.
- If we can't find any non-zero counters that were previously zero, it means the fuzzer is hitting the same code that was already covered so is not making progress (in the sense of code exploring).

Generators follow certain constraints in order to produce valid transactions. Some of these constraints include:

- Using existing accounts (obtained from the ledger). If we create new accounts, the fuzzer keeps a copy of their keypair in the fuzzer state so that they can be used to sign account updates.
- Using fees larger than the minimum fee (not checked by the transaction application logic, but it is checked in the transaction pool, so it is enforced to prevent false positives).
- Respecting transaction size boundaries (these are not checked by the transaction application logic but they are checked in the transaction pool, so we enforce them to prevent false positives).
- Signing transactions correctly (also not checked by the application logic, done in the transaction pool).
- In case of changing an account's balance, the fuzzer will try to "move" these funds from/to other accounts to keep the "excess fee" to zero (this is only needed for MINA transactions, in Custom Tokens this check is not done by the protocol and every zkApp has to use their custom logic)

The generated transactions are tested in two passes. :

1. The first pass calls the transaction pool logic, which performs many of the tests explained in my previous comment (minimum fee, size boundaries, signature). If this pass fails, then the transaction is discarded and a new transaction is generated. If the check passes then the transaction is stored in a ring-buffer in the fuzzer's state (called `cache_pool`) and it goes to the second pass.
2. The second pass calls the transaction application logic. Transactions that are fully applied in this pass (they are totally applied by the transaction application logic without any errors) are stored in a separated ring-buffer in the fuzzer's state (called `cache_apply`).

These caches are used to keep a fuzzing corpus (produced by the generators).

a) On each iteration the fuzzer will first try to get some random element from the `cache_apply` (with 0.9 probability), mutate it and send it to the two passes (pool logic and application logic).
b) If the fuzzer can't find any elements in the `cache_apply`, it will try to get an element from the `cache_pool` (with 0.5 probability), and send it to the two passes.
c) Otherwise, the fuzzer will call the generator to produce a new transaction (and send it to the two passes).

Mutators will use an existing transaction (from the corpus) and make one or more (usually a small amount) of changes to the transaction. Then the whole transaction is re-hashed and re-signed (otherwise every modified transaction would be invalid)

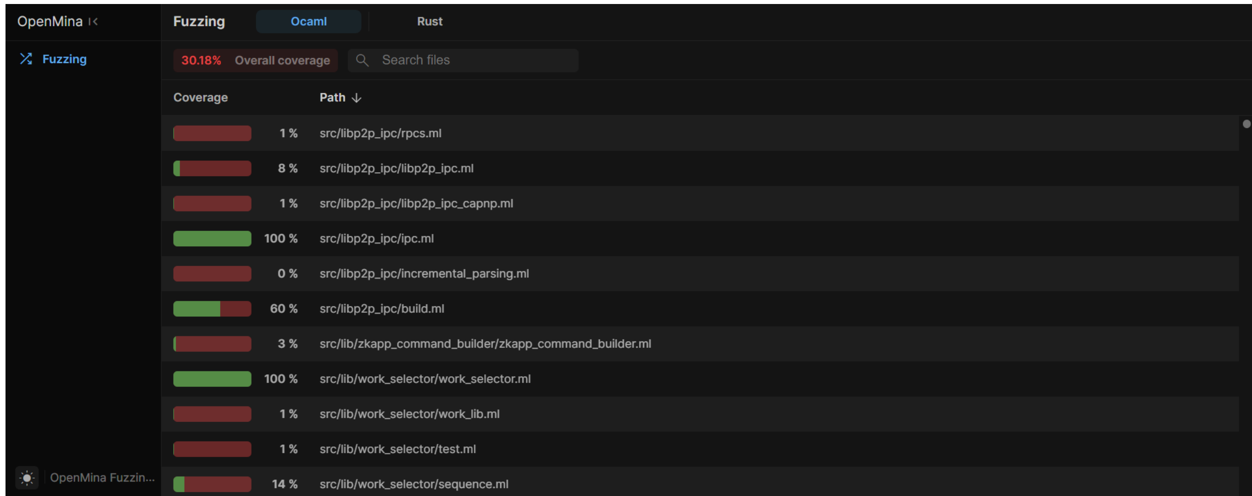
If the fuzzer is still making progress, we take a *snapshot* of the state of the ledger (and some extra fuzzer state). We keep multiple snapshots of the ledger state at different points so that we may revert the ledger state back in case the fuzzer is not making any more progress (no new coverage after a high number of iterations).

For instance, it is possible that we have populated the ledger with too many accounts, and there are not enough funds between them to pay for transaction fees. In that case, we will randomly select one of the saved snapshots and revert the ledger state to it (while still generating new transactions).

A visual overview of Mina's fuzzing

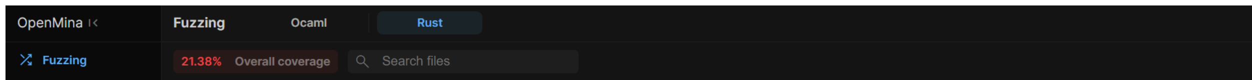
To visualize the process of fuzzing a Mina node, we have created a front end you can view via your internet browser.

Click on [this link] to open up the front end.



By default, the website loads up the OCaml tab.

Click on the Rust tab to open up the fuzzing overview for Mina's Rust code:



On both the OCaml and Rust tabs, we can see, from top to bottom:

Overall coverage - The overall percentage of the OCaml/Rust part of Mina's codebase that has been fuzzed. It is color coded, with red for under 50% fuzzed, yellow for under 80% fuzzed, and green for over 80% fuzzed.

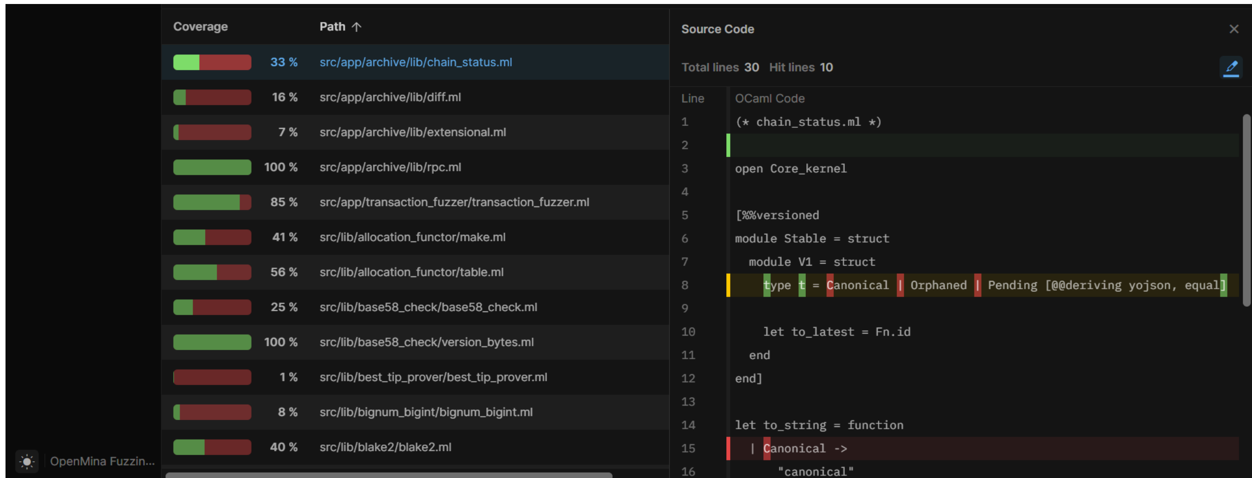
Search Files - type in a file name or file path you want to filter out in the list below in real time.

Below is a list of files of the OCaml/Rust part of the Mina codebase. On the left is their coverage percentage, both visualized as a bar and with a percentage next to it. The column to the right displays the file path for that file. Entries are updated in real time.

Clicking **Coverage** will sort these entries by how much percent they have been fuzzed in a descending order (from most fuzzed to least), clicking on it again will sort them in an ascending order.

Clicking on **Path** will sort entries alphabetically. Click on it again to sort in reverse alphabetical order.

Now click on an entry in the list of files to open up the sidebar.



The side bar displays the **Source Code** of the selected file along with the number of **total lines** and coverage (which lines have been fuzzed) being represented by **hit lines**.

Coverage highlighting can be toggled on or off by clicking on the highlighter icon in the top right corner of the sidebar.

Each highlight piece of the code has a tooltip that shows the number of times the line was executed during the test run.
The sidebar is color-coded:

- A green highlight means the code was executed at least once on the entire line.
- Yellow means the code was executed at least once on a part of the line and zero times on another part.
- If the highlight is red, then the code was executed zero times.

The user can copy a permalink(similar to GitHub) to a specific line of code by clicking on the line number.

We thank you for taking the time to read this article. For more details on fuzzers used in the Mina node, check out our GitHub repo. If you have any comments, suggestions or questions, feel free to contact me directly by email. To read more about OpenMina and the Mina Web Node, subscribe to our Medium or visit our GitHub.