Privacy Considerations

Privacy is important.

Keeping information private is difficult.

Once information is leaked, it cannot be unleaked.

What can Aztec keep private?

Aztec provides a set of tools to enable developers to build private smart contracts. The following can be kept private:

Private persistent state

Store state variables in an encrypted form, so that no one can see what those variables are, except those with the decryption key.

Private events and messages

Emit encrypted events, or encrypted messages from a private smart contract function. Only those with the decryption key will learn the message.

Private function execution

Execute a private function without the world knowing which function you've executed.

danger Privacy is not guaranteed without care. Although Aztec provides the tools for private smart contracts, information can still be leaked unless the dapp developer is careful. This page outlines some best practices to aid dapp developers.

Leaky practices

There are many caveats to the above. Since Aztec also enables interaction with thepublic world (public L2 functions and L1 functions), private information can be accidentally leaked if developers aren't careful.

Crossing the private -> public boundary

Any time a private function makes a call to a public function, information is leaked. Now, that might be perfectly fine in some use cases (it's up to the smart contract developer). Indeed, most interesting apps will require some public state. But let's have a look at some leaky patterns:

- Calling a public function from a private function. The public function execution will be publicly visible.
- Calling a public function from a private function, without revealing themsg_sender
- of that call. (Otherwise themsg_sender
- will be publicly visible).
- Passing arguments to a public function from a private function. All of those arguments will be publicly visible.
- Calling an internal public function from a private function. The fact that the call originated from a private function of that same contract will be trivially known.
- Emitting unencrypted events from a private function. The unencrypted event name and arguments will be publicly visible.
- Sending L2->L1 messages from a private function. The entire message, and the resulting L1 function execution will all be publicly visible.

Crossing the public -> private boundary

If a public function sends a message to be consumed by a private function, the act of consuming that message might be leaked if not following recommended patterns. Seehere for more details.

Timing of transactions

Information about the nature of a transaction can be leaked based on the timing of that transaction.

If a transaction is executed at 8am GMT, it's much less likely to have been made by someone in the USA.

If there's a spike in transactions on the last day of every month, those might be salaries.

These minor details are information that can disclose much more information about a user than the user might otherwise expect.

Suppose that every time Alice sends Bob a private token, 1 minute later a transaction is always submitted to the tx pool with the same kind of 'fingerprint'. Alice might deduce that these transactions are automated reactions by Bob. (Here, 'fingerprint' is an intentionally vague term. It could be a public function call, or a private tx proof with a particular number of nonzero public inputs, or some other discernible pattern that Alice sees).

TL;DR: app developers should think about thetiming of user transactions, and how this might leak information.

Function Fingerprints and Tx Fingerprints

A 'Function Fingerprint' is any data which is exposed by a function to the outside world. A 'Tx Fingerprint' is any data which is exposed by a tx to the outside world. We're interested in minimizing leakages of information from private txs. The leakiness of a Tx Fingerprint depends on the leakiness of its constituent functions' Function Fingerprintsand on the appearance of the tx's Tx Fingerprint as a whole. For a private function (and by extension, for a private tx), the following informationcould be leaked (depending on the function, of course):

- All calls to public functions. *The contract address of the private function (if it calls an internal public function).*This could be the address of the transactor themselves, if the calling contract is an account contract.
-
    - All arguments which are passed to public functions.
- All calls to L1 functions (in the form of L2 -> L1 messages).* The contents of L2 -> L1 messages.
- All unencrypted logs (topics and arguments).
- The roots of all trees which have been read from.
- Thenumber
- of'side effects'
- :* #new note hashes
-
    - **new nullifiers**
-
    - **bytes of encrypted logs**
-
    - **public function calls**
-
    - **L2->L1 messages**
-
    - **nonzero roots1**

Note: many of these were mentioned in the"Crossing the private -> public boundary"section. Note: the transaction effects submitted to L1 isencoded but not garbled with other transactions: the distinct Tx Fingerprint of each tx can is publicly visible when a tx is submitted to the L2 tx pool.

Standardizing Fingerprints

If each private function were to have a unique Fingerprint, then all private functions would be distinguishable from each-other, and all of the efforts of the Aztec protocol to enable 'private function execution' would have been pointless. Standards need to be developed, to encourage smart contract developers to adhere to a restricted set of Tx Fingerprints. For example, a standard might propose that the number of new note hashes, nullifiers, logs, etc. must always be equal, and must always equal a power of two. Such a standard would effectively group private functions/txs into 'privacy sets', where all functions/txs in a particular 'privacy set' would look indistinguishable from each-other, when executed.

Data queries

It's not just the broadcasting of transactions to the network that can leak data.

Ethereum has a notion of a 'full node' which keeps-up with the blockchain and stores the full chain state. Many users don't wish to run full nodes, so rely on 3rd-party 'full-node-as-a-service' infrastructure providers, who service blockchain queries from their users.

This pattern is likely to develop in Aztec as well, except there's a problem: privacy. If a privacy-seeking user makes a query to a 3rd-party 'full node', that user might leak data about who they are; about their historical network activity; or about their future intentions. One solution to this problem is "always run a full node", but pragmatically, not everyone will. To protect less-advanced users' privacy, research is underway to explore how a privacy-seeking user may request and receive data from a 3rd-party node without revealing what that data is, nor who is making the request.

App developers should be aware of this avenue for private data leakage.Whenever an app requests information from a node, the entity running that node is unlikely to be your user!

What kind of queries can be leaky?

Querying for up-to-date note sibling paths

To read a private state is to read a note from the note hash tree. To read a note is to prove existence of that note in the note hash tree. And to prove existence is to re-compute the root of the note hash tree using the leaf value, the leaf index, and the sibling path of that leaf. This computed root is then exposed to the world, as a way of saying "This note exists", or more precisely "This note has existed at least since this historical snapshot time".

If an old historical snapshot is used, then that old historical root will be exposed, and this leaks some information about the nature of your transaction: it leaks that your note was created before the snapshot date. It shrinks the 'privacy set' of the transaction to a smaller window of time than the entire history of the network.

So for maximal privacy, it's in a user's best interest to read from the very-latest snapshot of the data tree.

Naturally, the note hash tree is continuously changing as new transactions take place and their new notes are appended. Most notably, the sibling path for every leaf in the tree changes every time a new leaf is appended.

If a user runs their own node, there's no problem: they can query the latest sibling path for their note(s) from their own machine without leaking any information to the outside world.

But if a user is not running their own node, they would need to query the very-latest sibling path of their note(s) from some 3rd-party node. In order to query the sibling path of a leaf, the leaf's index needs to be provided as an argument. Revealing the leaf's index to a 3rd-party trivially reveals exactly the note(s) you're about to read. And since those notes were created in some prior transaction, the 3rd-party will be able to link you with that prior transaction. Suppose then that the 3rd-party also serviced the creator of said prior transaction: the 3rd-party will slowly be able to link more and more transactions, and gain more and more insight into a network which is meant to be private!

We're researching cryptographic ways to enable users to retrieve sibling paths from 3rd-parties without revealing leaf indices.

*Note: due to the non-uniformity of Aztec transactions, the 'privacy set' of a transaction might not be the entire set of transactions that came before.

Any query

Any query to a node leaks information to that node.

We're researching cryptographic ways to enable users to query any data privately.

Footnotes

1. All txs should set the kernel circuit public inputs for all roots tovalid
2. ,up-to-date
3. nonzero values, so as to mask which trees haveactually
4. been read from. The Sandbox will eventually automate this (see thisissue
5.).↩ Edit this page

Previous Architecture Next Limitations