CS261: Computer Security October 9, 2017

Advanced Blockchain-Based Concepts

Instructor: Raluca Popa *Scribe: Nick Spooner*

In the last lecture, we saw one very specific use of the blockchain: as a cryptographically secure decentralized currency. Blockchain technology, however, is much more powerful, and has seen many alternative uses. We will discuss one of these today: smart contracts, and their implementation as the Ethereum protocol. We will also discuss Zerocash, a protocol which aims to provide a cryptocurrency with provable anonymity.

1 Smart contracts

The basic notion of a smart contract is a program which runs 'on the blockchain'. What this means is that generating a new block or verifying the integrity of a block entails executing the program, and so the smart contract is executed by every participant in the protocol. This public verification ensures the integrity of the execution.

We can think of Bitcoin itself as a kind of simple smart contract, where the program to be executed is something like 'if Alice holds at least n bitcoins, transfer n bitcoins from Alice to Bob'. Verifying a block entails running this program. Protocols like Ethereum provide much more flexible and expressive smart contracts, using simple scripting languages.

2 Ethereum

In this section we will describe the operation of the Ethereum protocol for smart contracts.

2.1 Ether

Underlying Ethereum is a currency called ether, which serves two purposes. The first is to incentivize participation. In any blockchain-based system, the integrity of the shared state is guaranteed by cryptographic proofs of work (usually finding an input which hashes to something 'nice'). This is crucial in guaranteeing that the 'honest majority' of computational power in the system retains control of the blockchain. Generating these proofs is inherently expensive, and so in order to incentivize the creation of new blocks, there must be some mechanism by which parties who do so are compensated, as in Bitcoin. Ethereum presents the additional challenge that the cost of generating a block depends not only on the proof of work but also on the complexity of any smart contracts which are executed as a result of the block's creation. This is resolved in the Ethereum protocol by associating a cost (in ether) with every step of computation, which is paid to the party that 'mines' that block. We will discuss the details of how this works later.

The other use of ether is as an internal cryptocurrency which can be manipulated using smart contracts. That is, contracts can hold some amount of ether and transfer it elsewhere (including to other contracts, a form of message passing). This allows for interesting applications such as verifiable lotteries and decentralized autonomous organizations; the blockchain ensures that all transactions generated by the contract are as specified in its source code.

2.2 Accounts

In Ethereum, the basic object of state is an *account*. Every account is associated with a 20-byte address and an ether balance. There are two different kinds of accounts: externally owned accounts,

belonging to users of the system (and controlled by the user's private key), and contract accounts, which are 'autonomous' accounts controlled by contract code. In addition to the address and ether balance, therefore, contract accounts have associated with them some code and some storage which can be accessed and modified by the code.

To create an external account, a user generates a public-private key pair. The address of the account is generated from (a hash of) the public key, and the account is created by sending a transaction to that address. To create a *contract*, the user simply chooses some unused address and sends a transaction to that address, where the code of the contract is included in the transaction data.

2.3 Transactions and messages

State changes are effected by *transactions* and *messages*. A transaction is generated by an external account; a message is generated by a contract. Accordingly, they have slightly different formats, but they serve roughly the same purpose. In particular, **when a transaction or message is sent to a contract account, it causes the associated code to be executed**. A transaction consists of the following fields:

- the address of the *recipient* of the message;
- the *amount of ether* to transfer;
- optional data, such as the code of a contract or arguments to a function;
- the *gas limit* V and the *gas price* P, which we explain later.

Finally, the sender signs the transaction using her private key and attaches the signature to the transaction. From the signature one can easily derive the public key, and hence the address of the sender. Transactions are broadcast to the network to be added to the blockchain.

Messages are similar to transactions, with a few key differences. Since the message comes from a contract rather than an external user, there is no need to broadcast them on the network: the sending of a message is part of the computation involved in adding the block. Accordingly, messages also do not need to be signed, because their validity is implied by the correct execution of the contract and the validity of the message which caused that contract to be executed. This chain terminates at a transaction, whose validity is guaranteed by its signature. Finally, the *gas price* field is inherited from the gas price of the originating contract.

2.4 Contract execution and gas

When a contract account receives a transaction or message, its code is executed. Before execution, $V \cdot P$ ether is subtracted from the sender's account, if their balance is sufficient. The computation then has V 'gas' available. Some amount is immediately deducted from V corresponding to the byte-size of the transaction. Then the computation begins; each instruction in the Ethereum virtual machine (EVM) has a cost in terms of gas. Instructions are executed until either the computation is finished or no gas remains. If the computation finishes with R gas remaining, the sender receives a refund of $R \cdot P$ ether, and the miner $(V - R) \cdot P$. Otherwise, the action of the contract is reverted, and the entire amount $V \cdot P$ is sent to the miner.

Gas is used to compensate miners for executing computation. It is also prevents denial-of-service due to malicious or buggy contracts: if a contract would run for a very long time, or loop forever (loops are permitted by the EVM), it will stop executing when its gas runs out, and the sender pays for all the computation steps which did run. This avoids the problem of having to determine how long a contract will run for, which is impossible for a Turing-complete language such as EVM bytecode.

The *gas price* captures the fact that the ether currency and the computation itself are different resources, whose costs depend on vastly different factors. For example, if ether drops in value, the gas price chosen should be increased in order to make participation in the protocol worthwhile.

When a contract sends a message to another contract, the contract receiving the message is treated as part of the same execution, using up the gas allowance of the sender. For example, suppose that contract A is executed with gas limit 1000, and uses up 100 gas before calling contract B, which uses up 300 gas. Then when B returns, the remaining gas is 600. Gas price is maintained when one contract calls another, which makes it easier for a miner to estimate the reward it obtains from adding this transaction to a block.

2.5 Example transaction

We trace the execution of a simple transaction. Let T be a transaction whose recipient is a contract C, with ether value 10, gas limit $V = 2000$ and gas price $P = 0.001$. The transaction also contains a pair `(2, "charlie")` in its data field (storing each element in a 32-byte block). C has code:

```
if !self.storage[calldataload(0)]:
        self.storage[calldataload(0)] = calldataload(32)
```

which given `(i, str)` in the contract data, stores `str` at position `i` in C's storage.

The transaction proceeds as follows.

1. Check that the transaction is well-formed and correctly signed.
2. Check if the sender's balance is at least $V \cdot P = 2$ ether. If so, deduct 2 ether from the sender's account.
3. Set the starting gas value $g := V = 2000$. Deduct a per-byte gas fee for the size of the transaction: say the transaction is 170 bytes long and the per-byte fee is 5 gas, then we deduct 850 from g. The remaining gas is 1150.
4. Subtract 10 ether from the sender's account, and add it to C's account.
5. Run the code, subtracting the cost of each instruction from g. This sets the contract's storage entry 2 to the string "charlie". Suppose this required 187 gas; we subtract this from g, leaving $g = 963$.
6. Add $g \cdot P = 0.963$ ether back to the sender's account and return the updated state.

3 Anonymous Bitcoin

The Bitcoin protocol provides some amount of anonymity: payments are conducted between addresses, and no identifying information about the owner of the address needs to be provided. However, all of the transactions that take place are public, and so it is possible to determine a lot of information about the owners of addresses just from the transaction history. Moreover, anybody engaging in a bitcoin transaction with you can know your entire transaction history. There are two primary reasons why this is problematic:

- **Privacy.** Users in the bitcoin network may not want their spending habits and account balances available to anybody with whom they transact.
- **Fungibility.** Every bitcoin has an associated history. This can make some bitcoins worth less than others: for example, bitcoins which were used for criminal activity might be seen as 'dirty' and worth less than 'clean' bitcoins. This is an issue because we would like every bitcoin to have the same value, as with any currency.

3.1 Mixnets

One approach to privacy for Bitcoin is the 'mixnet'. This is a pool of users who shuffle transactions among themselves in order to hide, to some extent, the source and destination of each individual transaction. There are multiple problems with this scheme. The first is that it requires some trust in the mix operator: it is possible for the mix to trace or even steal bitcoins. This can be somewhat mitigated by chaining multiple independent mixes. Mixnets are still susceptible to statistical inference attacks, and do not achieve fungibility; indeed, it may be that you receive a bitcoin out of a mixnet which is less valuable than the one you put in.

3.2 Zerocash

Zerocash aims to solve the privacy problem in a provable way. The scheme uses succinct zero knowledge proofs (zk-SNARKs) to guarantee both privacy and integrity simultaneously. We describe a very simplified version of the protocol. To create a coin, the user generates a random serial number s and a trapdoor r, and computes a commitment $c_r(s)$ to s. He then sends a 'mint transaction' to the blockchain, containing $c_r(s)$ (but not s or r). The mint transaction will be processed only if the user pays, say, one bitcoin to some escrow service (which is verified by the miner). The value of zerocoins is thus dependent on Bitcoin.

To spend a coin, the user sends the serial number s of the coin to the blockchain, along with a zero knowledge proof of the assertion "I know r such that $c_r(s)$ is on the blockchain". The spend transaction succeeds if s was not part of any previous spend transaction (preventing double spending). If the spend succeeds, then the user is paid one bitcoin by the escrow service. Observe that because s is private when the coin is minted, and $c_r(s)$ is private when the coin is spent (by the zero knowledge guarantee), it is not possible to link the transaction which minted the coin to the one which spent it.

This scheme only allows users to mint and spend single coin units, and does not demonstrate how to transfer coins to other users. The Zerocash protocol provides a 'pour' operation, which allows coins to have variable denominations, and ensures that once a coin is transferred from one user to another, the sender cannot spend the coin. The details of this operation are beyond the scope of this summary.