Using Ethereum Smart Contracts to Democratize Charities

Jessie (Xueshan) Bai, William Sun, Stanley (Xi) Wang, Elaine Wong

May 6, 2018

1 Problem Statement

At present, most charities work by soliciting donations from the public to put towards a general cause, such as saving wildlife or helping people who are homeless or in poverty. However, using this framework, it is difficult for donors and potential donors to tell how the funds of the charity are being used: they could be going towards administrative costs, projects, advertisement, or other categories. This opacity results in **decreased motivation** for donors, as donations can feel low-impact if a donor will not know what their money is going towards.

2 Motivation

As a result of the aforementioned problem of decreased motivation caused by lack of transparency, we aim to create a **transparent, weighted voting framework** for charitable giving. Our is thus motivated by the following goals:

- Allowing donors to have direct input on what their money is used for
- Allowing donors to have more influence with higher donations
- Decentralization, to ensure the integrity of the democratic process
- Transparency, to guarantee that donors know where their money is going

It is easy to see that in the current framework for charitable giving, where donors simply click a general "donate" button on a web page or deposit money into a box in front of a shopping mall, the first two elements goals are not satisfied. Furthermore, transparency is damaged by the centralization of current charities: if the charity does choose to release a spending or budget breakdown for donors to look at, there is no way to validate or enforce this budget. Thus, decentralization is crucial to reassuring donors of transparency and integrity as we move charities towards a democratic process.

3 Existing Solutions

In our research, two main solutions that appeared were **Giveth**, which is in its alpha testing stage, and **Alice**, which is in its whitepaper stage. Both aim to build a more transparent framework for charitable giving. We will discuss these solutions and their shortcomings in terms of our goals below:

3.1 Giveth

Giveth is an Ethereum smart contract application for charities whose goal is to make charities' spending transparent to their donors. Their goal is to allow donors to follow their donations in real time and to have ownership over their donations, which means that donors continue have to control over what their money is used for after they donate. This is done by letting donors know exactly who is going to receive their money as well as giving donors the possibility to contact all involved parties directly. If the donor does not like what their money is being used for, they can retract their donation. If they do not use this veto power, the donor's money eventually becomes locked for a specific purpose. At that point, the money is put into a "Vault" and cannot be withdrawn by the charity unless the charity meets a predetermined milestone.

The structure of Giveth begins with three building blocks: DACs (decentralized altruistic communities), campaigns put together by DACs, and milestones, which are checkpoints in a campaign. A donor can donate directly to a campaign, or pledge to a DAC who will allocate their funds for them. A donor can then revoke their donation at any point up until their money becomes locked into a campaign.

Giveth satisfies most of the goals we mentioned in section two. However, one issue with Giveth is that donors do not have direct input that explicitly scales with the size of their donation. They can withdraw funds if they do not like the campaign it's being put into, but they cannot necessarily choose future campaigns that they would like to put their money into. Thus, they do not exactly have a direct positive influence on the future of the charity and use of their money if no existing campaigns align with their interests.

3.2 Alice

Like Giveth, Alice also attempts to make charities' spending transparent to their donors. They also follow the same model as Giveth, where donor's funds cannot be accessed by a charity unless it is verified that they have completed a certain milestone. Alice also allows donors to see exactly how much of their donation has been collected and which goals their donation paid for. Funds are tracked with Ethereum smart contracts.

Another feature Alice offers is the ability to vote on PUPs, or project update proposals. These are updates to existing projects that donors can either accept or reject, with their vote weighted by their donation amount.

However, likes Giveth, Alice does not provide a way for donors to give direct feedback or input on future projects. While Alice gives donors a say in the future of current campaigns by giving them input on how those campaigns change, we believe the donor's influence should still be more direct.

4 Our Solution

In light of these shortcomings, we have created a platform that allows donors to vote on future projects for the charity. In addition to giving donors direct input, we also introduce a motivational element of competition by allowing donors/voters to see the amount of votes for each projects. This would encourage people who are passionate about a certain voting option to donate. This can be used as an augmentation of the Giveth and Alice platforms, so that donors can track their funds after voting. Our implementation is detailed below.

4.1 Implementation

The best, most comprehensive description of our project's implementation can be found at the Github repository at the end of this report. However, here is a flow of how the contract could be used:

1. The administrator creates the contract.
2. Some donors donate to the contract because they are interested in the idea of a contract-based charity.
3. The administrator adds some voting options for donors to vote on.
4. The administrator locks in the options, formally starting the round of voting and setting a time at.
5. Other donors now know all of the options that could be voted on, and donate to vote for their favorite option.
6. Voting ends.
7. Any individual can call to "disperse", causing the contract to calculate the winner of that round of voting, and transfer its entire balance to the winner.

4.2 Security Concerns

1. **Charities voting for themselves.** If this contract is implemented naively, a charity could maliciously vote for themselves in order to win the ether donated by legitimate donors. For example, if a total of 100 ether has been donated and voted with by legitimate users, a malicious charity could potentially donate 101 ether to the contract and vote for themselves. If there were no more voting done, they would win back the 101 ether they donated and the 100 ether the legitimate users donated, effectively stealing the 100 legitimately donated ether and ruining the integrity of the voting system.

   We have brainstormed some possible designs to counter this:

   (a) Capping the voting power of each donor, or setting some kind of vote decay.

   Issue: This is exploitable by a sybil attack, with an adversary splitting their donations and votes across multiple wallets.

   (b) Allow multiple charities to be assigned for each cause, and randomly pick one of them. The charities should all be towards the same cause.

   Issue: Not as straightforward. There could still be coalitions of charities, and it may still be advantageous on average to vote just enough for yourself to tip the winning in your favor.

   (c) Mask who is voting for what, possibly until voting is over.

   Issue: This reduces the transparency of the charity, which is a key feature as it encourages competition and therefore more donations amongst honest donors.

   Even if we only make the votes cryptographic comments (as Abhi suggested after the presentation), due to the way that smart contracts are publicly inspectable, an adversary would at the very least be able to see the total weight of the votes that has already been voted with (but not necessarily the current vote counts). This solution is likely be the best way to combat this attack without harming the core ideals of the contract.

   (d) Not allow all of someone's contribution to be immediately used for the next vote. Thus, if set up correctly it could be economically inefficient to vote for yourself in many situations. This is the solution we implement, as an option, in our contract.

   Issue: This requires donors to trust the administrator to pick good charities in the future rounds. See the issue with centralization for

more details.

2. **The centralization of the contract.** At the moment, the contract is rather centralized, depending on a single administrator. While the administrator cannot realistically embezzle funds, they can pick bad charities or not pick any voting options at all, rendering donated ether unusable.

   Under a simple model, this may not be a significant issue, since people can donate after all of the voting options are established. However, if we want to stop certain exploits, solutions such as (d) above require more centralization and trust.

3. **Integer Over/Underflow**

   (a) Integer Overflow:
   This security concern is about the case when a balance reaches the maximum uint value (2^{256}), it will circle back to zero.

   Solution:
   The common address to this problem is to check if the sum of current balance and the amount transferred is actually greater than the current balance (i.e balanceOf[_to] + _value >= balanceOf[_to]).

   However, this should not happen in our case, since the amount of donation can hardly has an opportunity to approach such a large number.

   (b) Integer Underflow:
   This security concern is about the case when a minus 1 calculation is implemented in one of the functions. For example a operation like this: bonusCodes.length–;

   Solution:
   In our implementation we just avoid such case of decreasing value. The algorithm always clears the donated() to zero instead of minusing the amount donated (Underflow).

4. **Block Gas Limit Attack:**

   Attack that makes the block running into the block gas limit. Each Ethereum block can process a certain maximum amount of computation. If you try to go over that, your transaction will fail.

   Solution:

(a) The number of charities is created and therefore limited by the creator of the contract.

Our algorithm accomplish this by implementing two checks for the addVoteOption() function, the first checks if the person calling this function is the administrator/creator (Only Creator is allowed to add vote options); the second checks if the voting is already started (Can only add options before the voting starts and the admin controls the starting time). Therefore the number of options added is controlled by the administrator/creator and would not encounter gas attack related to this function.

(b) Donating to our account does NOT cost gas.

In our algorithm, only two functions (addVoteOption AND startVoting) are called by the administrator/creator; therefore only these two functions can cost gas from the creator account. However, since we limit the amount of options added (described in (a)) and startVoting is a one time function to be called, we avoid gas attack related to administrator/creator's account.

All the rest of the functions like disperse, donate, vote would most likely be called by the donors or some non-administrative people (though administrator could technically call them, they are unlikely going to do so). Therefore these functions will only cost gas from them instead of the creator account.

(c) There is no refund function in the contract so no possibility for malicious request.

An adversary could maliciously attack the admin by repetitively donating some small amount of ether and then requesting the refund. Though the donation would cost gas from the adversary, the refund function will inevitably transfer certain amount of ether from the admin to the adversary, therefore costing gas from the admin. In order to simplify the process and avoid such attack, we decide not to include the refund function in our contract.

5. **Reentrancy:**

Similar to the malicious refund calling attack, Reentrancy attacks the bug that involves functions that could be called repeatedly, before the first invocation of the function was finished. This may cause the different invocations of the function to interact in destructive ways, such as withdrawing the balance over and over again before the balance is eventually set to zero at the end of the program.

Solution:

(a) Set balance to Zero before transferring the money

Since the algorithm sets the user's balance to 0 in the function before the donation is added into a certain charity vote, no second (and later) invocations will succeed, and the donation would not take place over and over again.

(b) Use transfer() function

This is more of a problem in the older version solidity: picking between send() instead of call.value()() to prevent any external code from being executed. Solidity later introduces the new transfer() function; it is even safer than send() to prevent reentrancy.

6. **Timestamp Dependence:**

   This attack concern is mainly about the case when timestamp of the block is manipulated by the miner.

   Solution: Implement our own time tracking Algorithm

   In this case we implement our own time tracking algorithm. Specifically, only the administrator/creator can start the voting and the duration of voting time would be set at the same time. All the functions depend on the status of the voting (before voting/ during voting/ after voting), which would be checked in functions like vote(), addVoteOption(), startVoting(), isVotingActive() and disperse(). Thus we successfully avoid such timestamp attack.

5 Links

Github Repository

Demo from Presentation

6 Conclusion

A big thanks to Abhi and the TAs for the great semester. Please let us know if you have any further questions about the project and/or its setup!

7 Works Cited

- "Giveth." *Giveth*, Giveth, giveth.io.

- "Alice: Blockchain for Good." *Alice: Blockchain for Good*, ALICE SI Ltd, 2017, alice.si.