The Operational Cost of Ethereum Airdrops

Michael Fröwis and Rainer Böhme

Department of Computer Science, Universität Innsbruck, Austria

Abstract. Efficient transfers to many recipients present a host of issues on Ethereum. First, accounts are identified by long and incompressible constants. Second, these constants have to be stored and communicated for each payment. Third, the standard interface for token transfers does not support lists of recipients, adding repeated communication to the overhead. Since Ethereum charges resource usage, even small optimizations translate to cost savings. Airdrops, a popular marketing tool used to boost coin uptake, present a relevant example for the value of optimizing bulk transfers. Therefore, we review technical solutions for airdrops of Ethereum-based tokens, discuss features and prerequisites, and compare the operational costs by simulating 35 scenarios. We find that cost savings of factor two are possible, but require specific provisions in the smart contract implementing the token system. Pull-based approaches, which use on-chain interaction with the recipients, promise moderate savings for the distributor while imposing a disproportional cost on each recipient. Total costs are broadly linear in the number of recipients independent of the technical approach. We publish the code of the simulation framework for reproducibility, to support future airdrop decisions, and to benchmark innovative bulk payment solutions.

Keywords: Airdrop, Bulk Payment, ERC-20, Token Systems, Ethereum

1 Introduction

Fungible virtual assets, such as cryptocoins and tokens residing on a blockchain, are network goods: their value lies in enabling exchange. A coin is worthless if nobody else uses or accepts it. Its value grows quadratically in the number of users, according to Metcalfe's law; and still super-linear under more conservative theories [7]. As a result, new coins have to reach a critical mass until positive feedback sustains rapid growth [6].

This observation is taken to heart in the marketing of new coins. The community has adopted the term *airdrop* for the subsidized (often free) provision of new coins to selected lead users, typically holders of competing coins, with the intention to raise popularity and reach critical mass. Similar strategies are well understood in the economics [15] and marketing literature [13]. Whether and under which conditions airdrops are successful for cryptocoins and tokens are empirical questions that future work should tackle. Here, we study the operational costs of airdrops on Ethereum, the most popular platform for token systems.

Airdrops incur costs in the form of transactions fees paid to miners, which are shared between the initiator of the airdrop (often the developer or maintainer of a token, henceforth *distributor*) and the *recipients*, depending on the technical approach chosen by the distributor. The platform charges fees for instructions, space on the blockchain, and the size of the state information. The costs are not negligible because every recipient identifier contains cryptographic material with high entropy that must be communicated in the airdrop. Typically, the identifiers are included in the transaction payload and thus occupy space on the blockchain. Another difficulty faced by Ethereum airdrops is the lack of a bulk transaction method in the popular ERC-20 standard [1] for fungible tokens. This adds overhead due to repeated communication.

We have observed several solutions and workarounds to these problems in the wild, and synthesize our findings into the—to the best of our knowledge—first systematic overview on the technology behind airdrops on the Ethereum platform. We implement selected techniques in model smart contracts and measure their cost and resource consumption as a function of the number of recipients by executing the contracts in a simulated Ethereum node.

The rest of the paper is organized as follows. Sections 2 presents technical options for carrying out airdrops on Ethereum, including a discussion of the relevant parameters and necessary prerequisites. We distinguish push and pull approaches, internal and external batching, and revisit pooled payments. Section 3 presents the cost estimates from the simulation study in units of Ethereum's internal fee model (gas), the best level of analysis for comparison between options. Section 4 interprets the main findings in units of fiat currency (USD), the level of analysis that matters for business decisions. Section 5 connects to relevant related work, before we give an outlook and conclude in Section 6. Technical details of the simulation framework are placed in two appendices.

2 Technical Aspects of Ethereum Airdrops

This section gives an overview about technical considerations when conducting airdrops on Ethereum. We briefly discuss parameters of an airdrop chosen by the business side, then explain shortcomings of the default token transfer interface and the resulting technical workarounds. We apply the lens of operational costs, which in case of the Ethereum platform translates to estimating transaction fees in units of gas, the most comparable metric.

2.1 Parameters of an Airdrop

Before carrying out an airdrop, a couple of parameters need to be decided. One of the first things to decide on is *who* shall receive tokens. This is often done by a simple sign-up system (using e. g., Telegram, web forms, etc.), or by defining a measure of relevance on existing addresses to select the set of recipients. Companies like *Bounty One*[1] offer matching between airdrop- distributors and

[1] `https://bountyone.io/airdrops`, [Online; accessed 18 Jun 2019].

recipients as a service. The rationale behind this is simple: you prefer to hand out tokens to active users participating in the ecosystem, instead of sending them to inactive accounts. For example, the banking startup *OmiseGO* conducted one of the early Ethereum airdrops [18]. They used account balance as an activity indicator and simply handed out tokens to every address holding more than 0.1 Ether (ETH) at block height 3 988 888. This airdrop serves as good running example because all code, including a documentation of the rationales behind design decisions, is publicly available [2]. The threshold of 0.1 ETH is a very simple metric. It does not consider essential aspects, such as: are those accounts still active and able to use the tokens?[2]

Two other important parameters of an airdrop are the number of tokens to be dropped and their distribution over recipients. For example, OmiseGO dropped 5% of the total supply of OMG tokens. The distribution between recipients can be uniform or depend on properties of the recipient. OmiseGO allocated tokens to recipients proportional to their ETH balances at a specified point in time.[3].

Finally, the technical approach of how to transfer the tokens to the recipients needs to be defined. This involves choosing the software implementation to distribute tokens in bulk. This decision heavily depends on the existing infrastructure of the underlying token system.

In summary, the main decisions to be taken are:

- Who receives tokens (based on what metric)? *(recipient selection)*
- How many tokens per recipient? *(distribution)*
- Which technical approach to use for distributing the tokens? *(implementation)*

Although recipient selection and the token distribution strategy involve technical aspects, they are mainly driven by business considerations. Those are out of the scope of this work. Here, we focus on technical implementation options and their associated cost. Our results are an important input to the business decision because they quantify the operational cost of an airdrop.

2.2 Shortcomings of Vanilla ERC-20 Airdrops

ERC-20 [1] is the most prominent standard on the Ethereum platform today. It defines an API for token systems that enables: (1) the encoding of token properties,[4] (2) access to the balances of owners,[5] and (3) the transfer of tokens between accounts.[6] Moreover, ERC-20 defines logging and event notification.[7]

[2] This depends on the account type and state. For example, disabled contracts or contracts that are not programmed to transact with token systems will never be able to use the funds. This is also noted in [2].

[3] See `https://github.com/omisego/airdrop/blob/master/processor.py`, line 77

[4] Functions: `symbol`, `name`, `decimals`, `totalSupply`

[5] Functions: `balanceOf`

[6] Functions: `transfer`, `transferFrom`, `approve`, `allowance`

[7] Events: `Approve` and `Transfer`

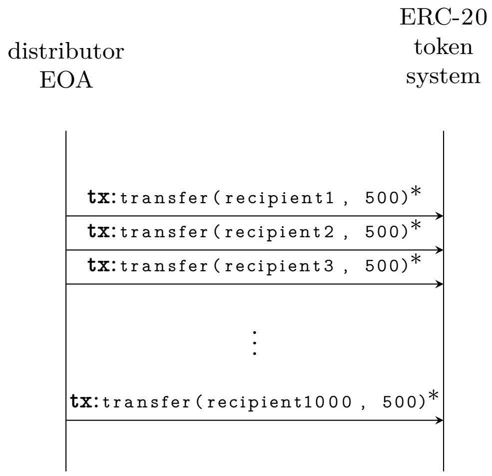

Fig. 1: Naïve push-style airdrop. One transaction per recipient from distributor to token system. The * indicates that the method is part of the ERC-20 standard.

The first ERC-20 token systems emerged already in late 2015.[8] Airdrops, on the other hand, are a more recent phenomenon starting to gain traction in early 2018. As a consequence, the ERC-20 API does not include a batch transfer method to directly transfer tokens to *multiple* recipients in one transaction. The lack of this functionality in legacy token systems makes airdrops more expensive. The immutability of contracts, which is often a desired feature, prevents legacy token systems from adding batch capabilities after deployment.

Hence, implementing an airdrop in vanilla ERC-20 proceeds as shown in Figure 1. The distributor uses an externally owned account (EOA) in order to send one transaction for every recipient. Each transaction invokes the `transfer` method of the token system in order to update its internal state. The fixed cost per transaction is 21 000 units of gas, which are not recoverable and add to the overhead of this approach. To avoid issuing one full transaction per airdrop recipient, the community developed several optimizations to reduce cost and avoid network congestion.

2.3 Optimizations

We distinguish three avenues for improvements. Transaction batching helps to reduce communication costs. The pull approach shifts part of the burden to the recipient and potentially conserves tokens and fees from recipients who do not collect their share. Off-chain approval saves cost by avoiding to store the list of account identifiers on the blockchain. We discuss each of these avenues in the following subsections.

[8] The first ERC-20 token. Block: 490 326,
Address: `0xEff6425659825E22a3cb00d468E769f038166ae6`

2.3.1 Transaction Batching

The easiest way to save cost is by removing the overhead of issuing one transaction for each airdrop recipient. A cheaper alternative to transactions are *message calls* (also known as *internal transactions*) invoked by contracts. The difference in fixed cost (without payload) is substantial: a transaction costs 21 000 units of gas, compared to 700 for a message call. Message calls to the same contract are yet an order of magnitude cheaper and cost about 10 units of gas. Message calls are generated in a loop over the list of recipients, which is given as an argument to a single initial transaction.

Let n be the number of recipients, then the cost savings s are

$$s_1 = n \cdot 21000 - (n \cdot 700 + 21000), \tag{1}$$

if the loop is implemented in a different contract than the token system. This method is called *external batching* and visualized in Figure 2. To give an intuition for the source of the savings, recall that the batch is authorized by a single signature as compared to to one signature per transaction in the naïve approach.

Even higher savings of

$$s_2 = n \cdot 21000 - (n \cdot 10 + 21000) \tag{2}$$

are possible if the loop is implemented directly in the token system contract (*internal batching*). The difference between external and internal batching can be explained by the penalty of fetching new code from disk, which applies n times in the case of external batching. However, changing the token systems's contract may either be impossible because it is already deployed immutably, or not desired in the fear of introducing new bugs or losing investor trust.

The savings are upper bounds that are only achievable if all n identifiers fit into one transaction. The size of transactions is restricted by the block gas limit.[9] Larger recipient lists must be split into several batches, each incurring the fixed cost of one transaction.

2.3.2 The Pull Approach

As already mentioned, distributing tokens to inactive accounts or to recipients not interested in the token has little to no value to the distributor. This problem can be addressed during recipient selection, by

- evaluating appropriate technical indicators,
- collecting information provided in sign-up, or
- using the specialized services in the ecosystem who administer panels of potential recipients.[11]

We are not aware of any literature on the effectiveness of these options and consider it out of our scope.

[9] Gas limit at the time of writing is 8 000 029 in block number 8 014 738

[10] For example, `https://multisender.app/`, [Online; accessed 25 Jun 2019].

[11] `https://bountyone.io/airdrops`, [Online; accessed 18 Jun 2019].

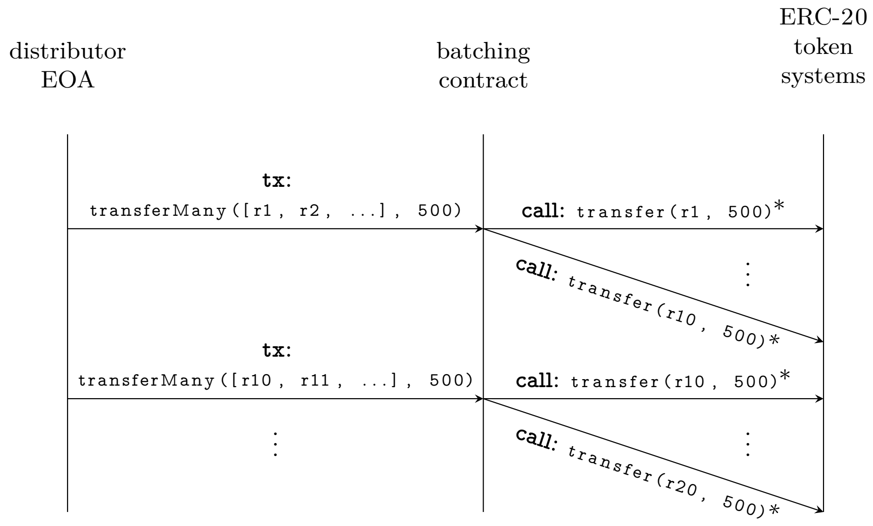

Fig. 2: External batching, push-style airdrop. The token system has **no** batching capabilities. Batching is done by an external contract (either own or service[10]). Note the calls instead of transactions.

A more technical approach is to condition the token transfer on on-chain user interaction. This type of *pull-style airdrop* can be implemented using the `approve` function of ERC-20. Instead of directly transferring tokens to the recipients during the airdrop, the distributor gives the recipient the right to withdraw the airdropped amount. The distributor may specify a deadline for the withdrawal and reclaim the remaining tokens thereafter.

Arguably, the additional effort and cost for the recipient ensures that the distribution is more targeted.

The pull approach has a couple of downsides. First, the cost for the recipient and the distributor are significant. Both sides pay about as much as the distributor pays for a push-style airdrop. This even holds when the distributor approves in batches as described in Section 2.3.1 (see Figure 3). Second, a known front-running attack against the ERC-20 `approve` logic requires the distributor to set all allowances to zero before updating them with new values [1, 20]. This approximately doubles his cost. Third, many existing token systems do not implement the `approve` method and therefore cannot use the pull approach [11]. Lastly, recovering the unclaimed tokens after the deadline costs about as much as transferring them.

2.3.3 Off-Chain Approval

The Ethereum community has realized that storing every recipient address on the chain leads to network congestion as well as to high cost. One approach that does not require to store all recipient addresses on-chain are *pooled payments* [5].

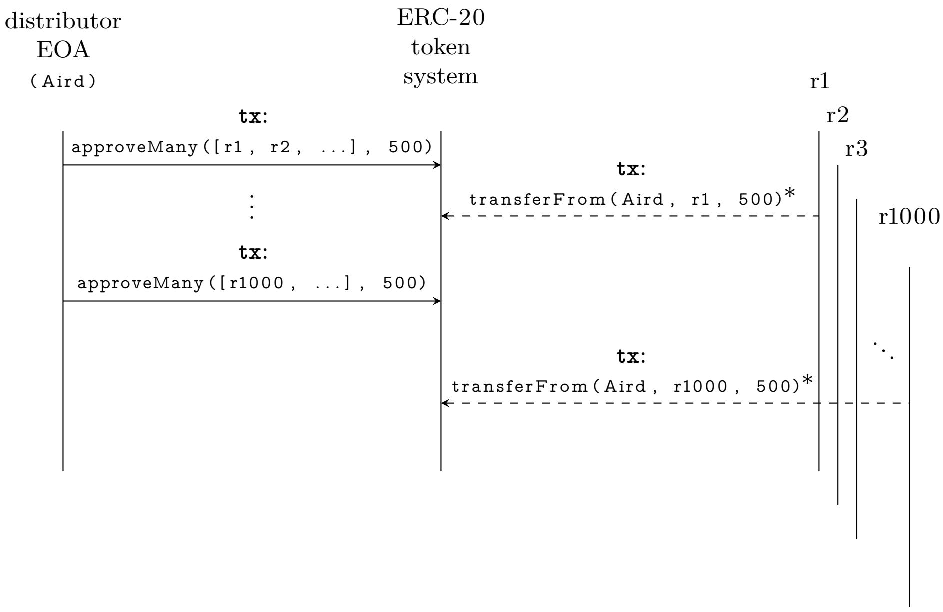

Fig. 3: Pull-style airdrop with non-standard `approveMany` for internal batching. Recipients have to interact with the ERC-20 contract to finally receive the funds. The distributor-side can be batched. Recipients must use individual transactions. Dashed lines mean recipient pays.

They are inspired by *Merkle mine* [4], an approach developed for token systems to define the initial allocation to a large number of owners.

Pooled payments resemble pull-style airdrops, as the distributor approves the recipient to withdraw a certain amount of tokens through a transaction. But pooled payments do not store the entire approval on-chain. Instead, the distributor encodes the list of recipients with denominations in a Merkle tree [16], where leafs are concatenations of recipient addresses and amounts. The approving contract has to store the root hash of the Merkle tree only (see Fig. 4, `approveReceivers(merkle_root)`). The list of recipients is published off-chain. To claim funds, every recipient needs the list and computes the Merkle tree as well as a Merkle proof for his entry. The recipient then sends a transaction to the contract with his address (implicit by the signature on the transaction), the amount, and the Merkle proof (see Fig. 4, `claimMerkle(500, merkle_proof)`). This allows the contract to compute the hash of the leaf node from the message sender (signature) and the amount in the argument. Using the Merkle proof, it verifies the correctness of the claim, checks its freshness, and transfers the funds. To prevent double-claiming, the contract must store a record of this withdrawal. Several methods exist to keep this as compact and cost-efficient as possible.

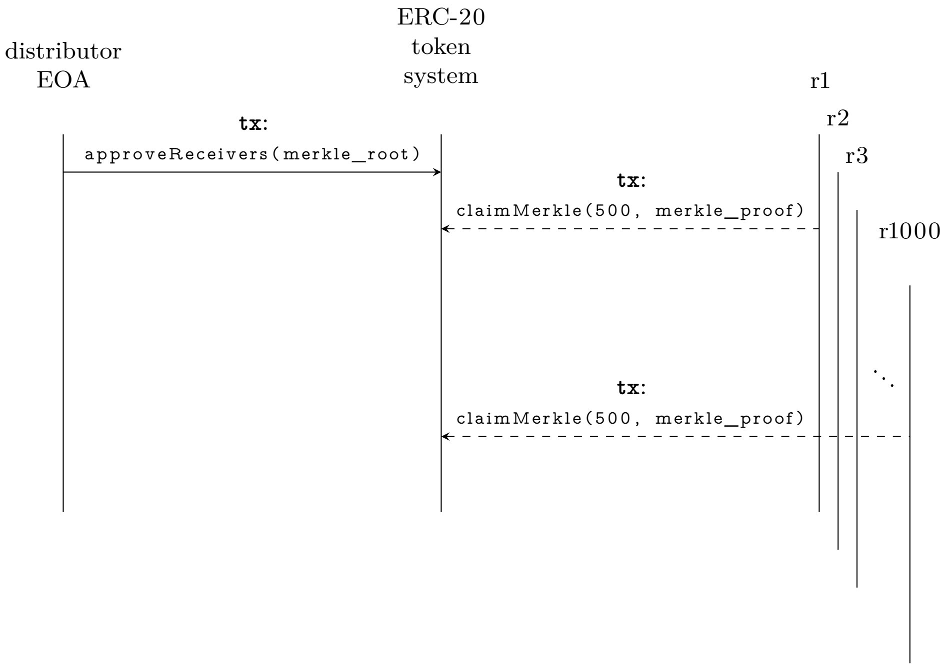

Fig. 4: Pooled payments: non-standard, internal, pull-style airdrop. Cost is constant for distributor. List of recipients is public.

Unlike for normal pull payments, the distributor has constant cost independent of the number of recipients. Most of the airdrop cost is shifted to the recipient.[12]

It is also worth mentioning that pooled payments based on off-chain approvals have some usability issues that may delay their adoption for airdrops. Both recipients and distributors need tools to do off-chain computation (Merkle tree, proofs) and data retrieval (list of recipients). We are aware of one business that seems to bet on the adoption of this approach.[13]

2.4 Miscellaneous Aspects

Optimizing the airdrop strategy and the code involved in the airdrop workflow are not the only things to consider when doing airdrops.

Gas Token: Gas Token[14] provides a way to pre-acquire gas in periods when gas is cheap. Those gas tokens can be "redeemed" when the gas price is high. The

[12] This accounts to: one transaction per recipient, Merkle proof verification, and storage of withdrawal record.

[13] The claim of constant distributor cost in the Coinstantine whitepaper indicates the use of pooled payments. See `https://www.coinstantine.io/`, [Online; accessed 22 Jun 2019].

[14] `https://gastoken.io/`, [Online; accessed 21 Jun 2019]

gas token mechanism exploits the fact that Ethereum refunds gas when storage resources are freed. To include this mechanism in airdrops, the functionality to redeem gas tokens must be built into either the token system or the batching contract. Gas tokens are already used in other areas, such as arbitrage bots [10].

Systemic Risk: Airdrops can be a systemic risk for the Ethereum platform, if not used carefully, since they use large amounts of resources (gas). Coindesk [3] reports that the uptake of airdrops in conjunction with questionable incentives set by the exchange FCOIN led to substantial network congestion, gas price increases, and wasted resources. Reportedly, OmiseGO also considered the impact of its airdrop on the network and decided to limit their batches such that they never use more than 50% of the block gas limit [2]. We adopt the idea of such a limit for the simulations in the following.

3 Cost Estimates

In the following we compare simulated costs of different airdrop techniques. The simulation gives us valid estimates of total cost, which puts the back-of-the-envelope calculations of savings in Eqs. (1) and (2) into perspective. The simulation framework, the constants used, a complete list of scenarios, and the code can be found in Appendix A and B. It can serve as starting point for facts-based airdrop decisions and to benchmark new solutions.

All 35 scenarios are combinations of the approaches discussed in Section 2.3. Table 1 resolves the labels used in the figures below.

Table 1: Approach labels and descriptions.

Label	Description
NAIVE:	No batching is applied. One transaction per recipient.
PUSH:	Push-style airdrop as discussed in Section 2.3.
PULL:	Pull-style airdrop as discussed in Section 2.3.
EXTERNAL_BATCH:	External batching as discussed in Section 2.3.
INTERNAL_BATCH:	Internal batching as discussed in Section 2.3.
UNIFORM:	One amount per batch. Otherwise n different amounts are sent.
RECIPIENT_COST:	Recipient cost of pull-style airdrop. All recipients claim funds.
BASE_LINE:	Baseline for pull style airdrop, see Appendix A.

Figure 5 presents the gas cost as a function of the number of recipients. We only show strategies viable when targeting a less than 50% block fill grade. Observe that all strategies behave broadly linear. The fixed cost per batch is negligible. NAIVE|PUSH and BASE_LINE|INTERNAL_BATCH|PUSH|UNIFORM|100 serve as upper and lower bounds and thus are benchmarks for the other strategies. The lower bound (baseline) simulates a push-style airdrop only considering communication and storage cost.[15]

Although, INTERNAL_BATCH|PULL|UNIFORM|100 appears to be the cheapest strategy, this is only true for the distributor. Recall that in pull-style airdrops recipients have to send additional transactions in order to withdraw their tokens. If we sum up the costs of the recipient (PULL|RECIPIENT_COST) and distributor, the pull-style airdrop is by far the most expensive, 32% more costly than NAIVE|PUSH.

The biggest improvement for both parties compared to the NAIVE|PUSH is archived by INTERNAL_BATCH|PUSH|UNIFORM|100, saving roughly 42%. The baseline suggests that savings up to 58% are be possible. If we compare internal vs. external batching, the internal strategies are save around 8% compared to their externally batched counterparts. The uniform strategies only save about 1% compared to their counterparts. The savings might go up if larger amounts are transferred, which require more non-zero bytes to encode. However, the batching contract could support logarithmic scaling or batch-wide multipliers, which make our approximation with two non-zero bytes per amount realistic again.

Figure 6 shows a single simulation run for 1000 recipients. Given the approximate linearity the number of recipients, this view is sufficient to compare the strategies. This time we present all strategies. We color code the minimal block fill grade in which each strategy gets feasible. The first thing to observe is that batches of more than 300 recipients are not feasible with current block gas limits[16]. Only the pull approaches and the baseline can manage a batch size of up to 300. Note that NAIVE|PUSH as well as the RECIPIENT_COST are feasible even with a threshold of 10% block fill grade, since no batching is applied.

We did not simulate the pooled payment strategy (see 2.3.3). Since the distributor cost is constant by only storing the Merkle root, this would make it the by far cheapest option for distributors. The recipients have to withdraw the tokens in a very similar manner to the PULL|RECIPIENT_COST strategy. In addition, the recipient has to pay for the verification of the Merkle proof and the storage of the withdrawal record.

4 Discussion

The above results show differences in gas consumption. More relevant units of operational cost for the distributor are ETH, if the distributor is already invested in Ethereum, and fiat (USD). To put our results into perspective, we estimate the cost savings per 1000 recipients in USD. This entails two conversions with variable rates: from gas to ETH and from ETH to USD. The first conversion is governed by the fee market mechanism and the miner's transaction inclusion strategy. Since transactions compete for inclusion in the chain, the gas price (in ETH) depends on the network load. The second conversion rate is found

[15] This rests on the assumption that other computation cost can be optimized. See Appendix A for more details.

[16] The block gas limit used as cutoff can be found in Table 2. Appendix A.

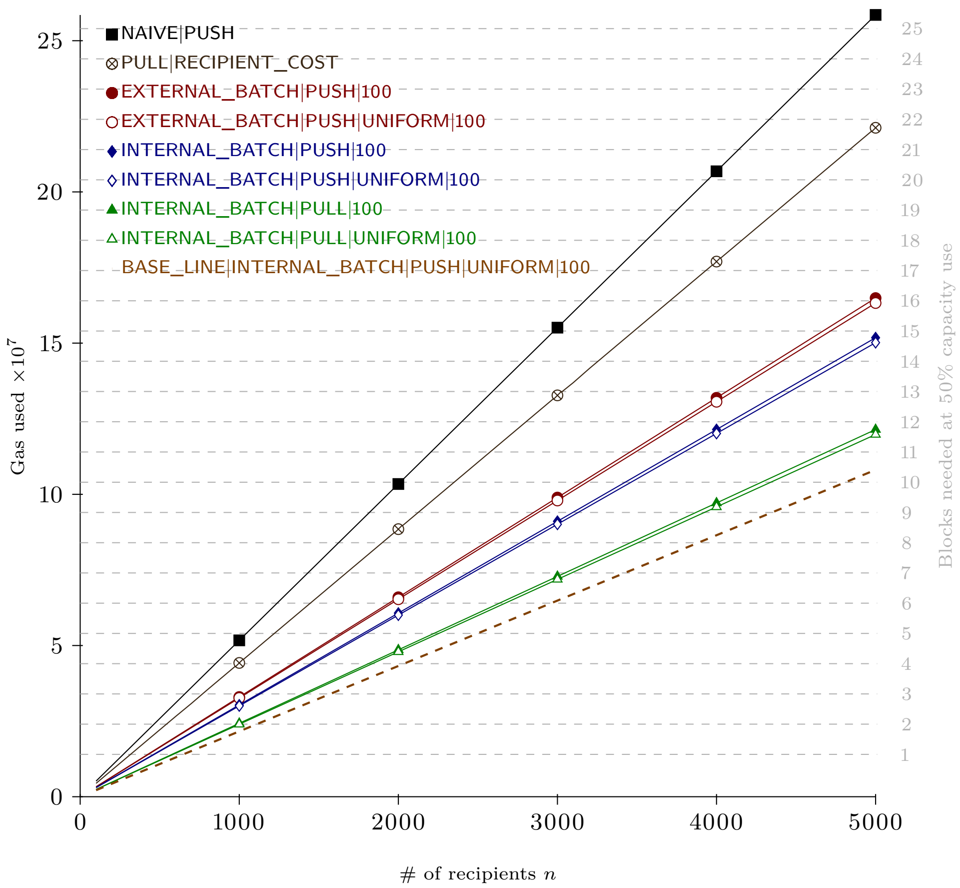

Fig. 5: Simulated cost for n recipients using different strategies. The chart shows the subset of strategies whose transactions fit into 50% of the block gas limit. Cost is not discounted, i. e., we assume all recipients are new to the token system, which makes the airdrop a bit more expensive.

on the cryptocurrency exchanges in the ecosystem. The price depends on demand and supply of cryptocurrency, which supposedly follow investors' economic expectations.

We do not aim to explain price formation in this work (although airdrops may affect prices in the short run), but take an empirical approach. Figure 7 shows the co-movement of both prices from January 2017 to June 2019 on a log-log scale. We calculate 60-day moving averages and represent each center day as dot, color-coded by the calendar month. The dashed lines connect levels of equal gas price in USD. Observe that both prices follow different dynamics, hence are not strongly coupled by a single market mechanism. While the ETH/USD exchange rate varies over two orders of magnitude in the sample period, the gas price in

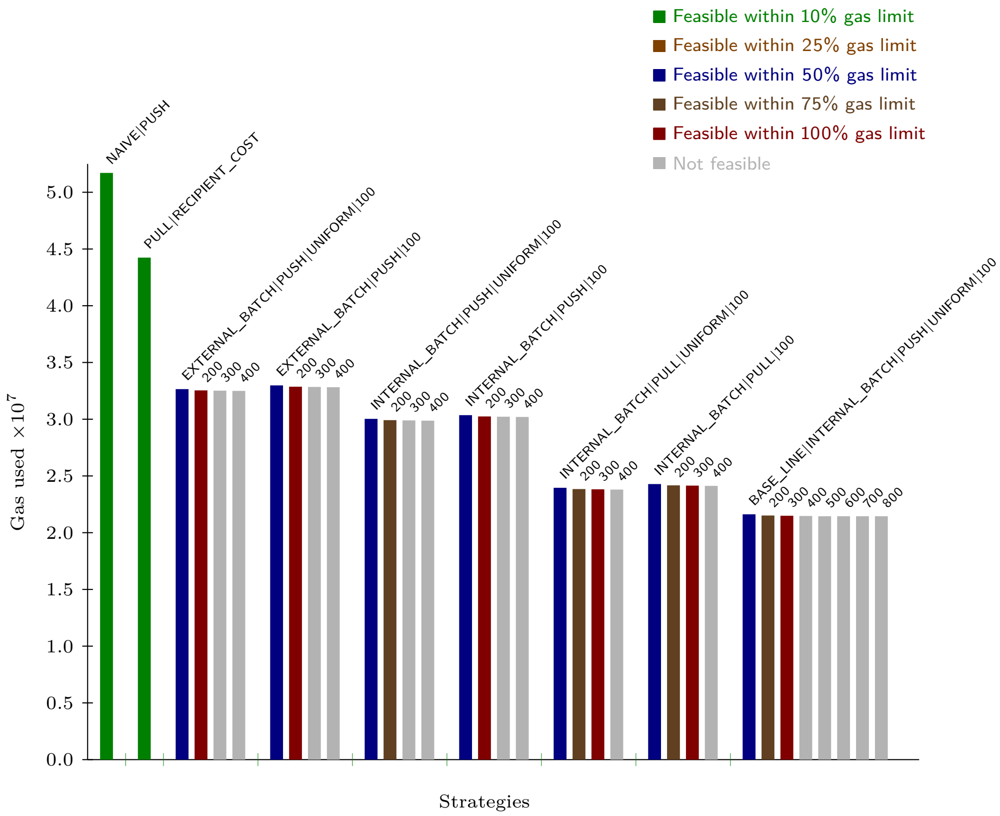

Fig. 6: Cost and feasibility of different strategies with 1000 recipients with different cutoffs on the block gas limit.

ETH remains in a much narrower band. However, it exhibits more sudden jumps, which relate to extreme values (e. g., due to congestion) that enter or leave the moving window. One can also speculate if the introduction of gas tokens in spring 2018 has narrowed the band of gas price movement due to the counter-cyclical behavior of gas token investors.

To get an idea of airdrop costs in USD, the dashed line marked with 15 cents (of USD) per recipient seems a good rule of thumb. This price level was applicable for a naïve push-style airdrop in May 2017, October 2018, and in March and May 2019. More efficient strategies have costed around 7.5 cents per recipient. Given the variability of both prices (some of which is hidden by the moving average), it seems that the right timing is at least as important as the strategy.

To continue the example from above, the OmiseGO airdrop distributed tokens to 450 000 recipients, using an externally batched push-style approach. By applying the gas-to-USD conversion rate indicated in Figure 7, we can estimate the cost at roughly 44 523 USD. Airdrops of this size occupy 50% of the available

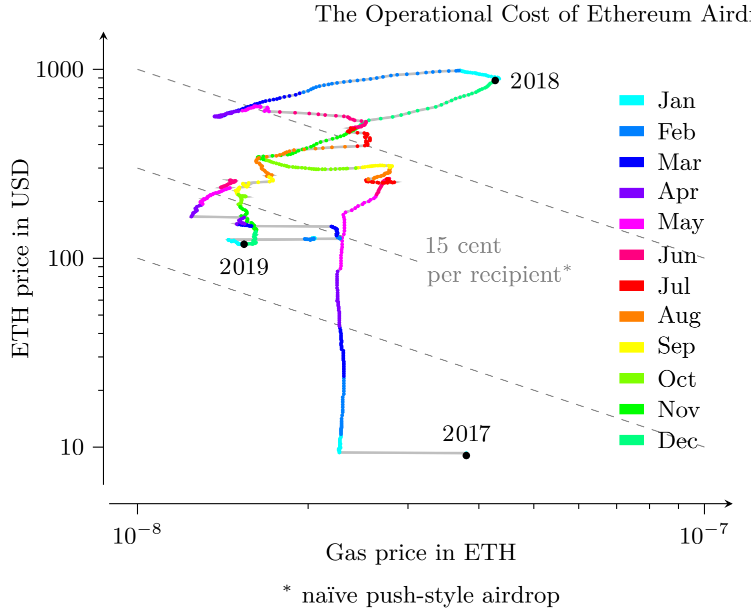

Fig. 7: Dynamics of the price of Ethereum resource use, broken down into components. The plot shows 60-day moving averages of daily prices reported by Etherscan.io. Dashed lines connect levels of equal gas price in USD.

capacity in 1440 blocks, taking at least 6 hours to complete. As a consequence, early recipients have an advantage when selling tokens on an exchange immediately after receipt. This suggests that token systems should support time locks in order to enable large and *fair* airdrops.

5 Related Work

Our work connects to prior works on the systematic analysis of token systems on the Ethereum platform, gas efficiency, and one seminal publication on airdrops.

Token Systems and ICOs: Howell et al. [14] study the success factors of 440 ICOs on Ethereum based on propriety transaction data, presumably acquired from exchanges and other intermediaries, and manual labeling. Their main interest is in the relationship between issuer characteristics and indicators of success. The regression analyses find highly significant positive effects on liquidity and volume of the token for independent variables measuring the existence of a white paper, the availability of code on Github, the support by venture capitalists, the entrepreneurs' experience, the acceptance of Bitcoin, and the organization of a pre-sale. No significant effect is found for airdrops.

Friedhelm et al. [19] study Ethereum token systems from a network perspective. They find that the degree distribution of the token network transfers does not follow a power law, but is dominated by a few hubs. In particular recipients of initial tokens mainly trade with these hubs. Some tokens systems seem to be very illiquid. Airdrops are not considered.

Gas Usage: Chen et al. [9] identify underpriced instructions (even after the 2016 gas price adjustment) and propose an adaptive pricing scheme. Their main interest is to raise economic barriers against congestion, which in the worst case enables denial of service attacks on the systemic level.

In a different work, Chen et al. [8] use pattern matching to identify code sequences that can be further optimized for gas use in smart contracts deployed until 2016. Naegele and Schett [17] pursue a similar goal with the help of SMT solvers. Both source report ample room for improvement. While the referenced works optimize on the instruction level, the optimizations studied in this paper primarily seek to minimize communication overhead.

Airdrops: Airdrops are a rather new topic. We are aware of one academic paper only. Harrigan et al. [12] raises awareness for privacy implications of airdrops when identifiers of one chain (*source chain*) are reused to distribute coins on another chain (*target chain*). Sharing identifiers between chains in general gives additional clues for address clustering.

To the best of our knowledge, we are the first to compare the gas costs of technical solutions for airdrops of tokens on the Ethereum platform.

6 Conclusion and Outlook

This work compared the efficiency of bulk transfer approaches on the Ethereum platform, a general problem that became particularly relevant with the uptake of token airdrops. It turns out that many of the approaches we systematized and reviewed are workarounds for architectural short-comings of the Ethereum platform or the popular ERC-20 standard for fungible tokens. The cost efficiency of the approaches differ roughly by a factor of two. Moreover, the most cost-efficient solutions for the distributor impose significant cost on the recipient, which might thwart the very intention of an airdrop as marketing tool. We release our simulation framework and the model contracts for reproducibility, as testbed for actual airdrops, and as benchmarking suite for new solutions.

The choice of approach is constrained by properties of the token system. This mainly relates to the penalty of repeatedly calling a method from *another* contract, which appears disproportional to the computational effort of the node. While a remote call is indeed expensive at first use, every repeated call is sped up through caching. Ethereum's gas price schedule seems to unfairly discriminate against bulk operations, an issue that designers of price schedules for future blockchain platforms should fix. On Ethereum as it stands, token issuers are best advised to reflect about airdrops before deployment of the token system contract. Planning ahead is vital in an environment where code cannot be amended easily.

While framed and motivated for the application of airdrops, our analysis generalizes to any kind of bulk operation on lists of incompressible items. Future designs of blockchain platforms should consider mitigating most of the issues discussed here by supporting a global index for constants with high entropy.

Some cryptographic material, in particular public keys and commitments, must be stored on-chain in order to enable authorization of actions by the knowledge of secrets. But if they do not double as references, as in Ethereum, then every value has to be stored (and paid for) only once. Furthermore, if all contracts have access to all public information on the blockchain, sets can be reused and storage space saved. In short, the community needs a DRY (don't repeat yourself) principle for data on the blockchain.

Acknowledgments

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 740558.

References

1. EIP 20: ERC-20 Token Standard. https://eips.ethereum.org/EIPS/eip-20 (2015), [Online; accessed 18 Jun 2019]
2. OmiseGO tokens airdrop. https://github.com/omisego/airdrop (2017), [Online; accessed 18 Jun 2019]
3. Ethereum's Growing Gas Crisis (And What's Being Done to Stop It). https://www.coindesk.com/ethereums-growing-gas-crisis-and-whats-being-done-to-stop-it (2018), [Online; accessed 18 Jun 2019]
4. MerkleMine Specification. https://github.com/livepeer/merkle-mine/blob/master/SPEC.md (2018), [Online; accessed 18 Jun 2019]
5. Pooled Payments (scaling solution for one-to-many transactions). https://ethresear.ch/t/pooled-payments-scaling-solution-for-one-to-many-transactions/590 (2018), [Online; accessed 18 Jun 2019]
6. Böhme, R.: Internet protocol adoption: Learning from Bitcoin. IAB Workshop on Internet Technology Adoption and Transition (ITAT), Cambridge (2013)
7. Briscoe, B., Odlyzko, A., Tilly, B.: Metcalfe's law is wrong. IEEE Spectrum 43(7), 34–39 (2006)
8. Chen, T., Li, X., Luo, X., Zhang, X.: Under-optimized smart contracts devour your money. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 442–446. IEEE (2017)
9. Chen, T., Li, X., Wang, Y., Chen, J., Li, Z., Luo, X., Au, M.H., Zhang, X.: An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks. In: International Conference on Information Security Practice and Experience. pp. 3–24. Springer (2017)
10. Daian, P., Goldfeder, S., Kell, T., Li, Y., Zhao, X., Bentov, I., Breidenbach, L., Juels, A.: Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. arXiv preprint arXiv:1904.05234 (2019)
11. Fröwis, M., Fuchs, A., Böhme, R.: Detecting token systems on ethereum. arXiv preprint arXiv:1811.11645 (2018)
12. Harrigan, M., Shi, L., Illum, J.: Airdrops and privacy: A case study in cross-blockchain analysis. In: 2018 IEEE International Conference on Data Mining Workshops (ICDMW). pp. 63–70. IEEE (2018)

13. Hill, S., Provost, F., Volinsky, C.: Network-based marketing: Identifying likely adopters via consumer networks. Statistical Science 21(2), 256–276 (2006)
14. Howell, S.T., Niessner, M., Yermack, D.: Initial coin offerings: Financing growth with cryptocurrency token sales. Tech. rep., National Bureau of Economic Research (2018)
15. Katz, M.L., Shapiro, C.: Systems competition and network effects. Journal of Economic Perspectives 8(2), 93–115 (1994)
16. Merkle, R.C.: A digital signature based on a conventional encryption function. In: Conference on the theory and application of cryptographic techniques. pp. 369–378. Springer (1987)
17. Nagele, J., Schett, M.A.: Blockchain superoptimizer. http://www.maria-a-schett.net/talks/2019_04-MaS_imdea.pdf (2018), [Online; accessed 24 Jun 2019]
18. nharrison: Airdrop update 2. https://steemit.com/ethereum/@nharrison/what-you-need-to-know-about-the-omisego-airdrop (2017), [Online; accessed 25 Jun 2019]
19. Victor, F., Lüders, B.K.: Measuring ethereum-based erc20 token networks. In: International Conference on Financial Cryptography and Data Security (2018)
20. Vladimirov, M., Khovratovich, D.: Erc20 api: An attack vector on approve/transferfrom methods. https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA_jp-RLM, [Online; accessed 25 Jun 2019]

A Simulation Environment

For the simulation, we use `ganache-cli`,[17] an implementation of an Ethereum node specifically designed for development and testing of smart contracts. The Ethereum virtual machine of `ganache` builds on `ethereumjs`. We implement the simulation in Javascript and run it on `nodejs` version `v8.15.0`. The `web3` library serves as interface to the node.

Every simulation run deploys a fresh versions of the simulated contracts and generates new accounts for all recipients as well as the distributor. Consequently, each transfer requires a new storage slot for the recipient. This costs 20 000 gas, in contrast to 5000 if an additional token is transferred to a recipient who already owns tokens of that type. To distinguish these two cases, we calculate a **discounted** scenario by subtracting the gas cost difference of 15 000 per recipient. We report the discounted results in Figure 11 for completeness. This scenario to some extent contradicts the purpose of an airdrop, which is to distribute tokens to new owners.

In all scenarios we assume a two-byte number of tokens to be distributed to all recipients. This is relevant because the number of zero bytes in the transaction input influences the gas cost. If the number of tokens per recipient requires more than two bytes, the cost gap between uniform and non-uniform distribution grows. External and internal batch transfer functions can reduce the cost of non-uniform distributions of highly divisible tokens by implementing a batch-wide amount multiplier. We have not measured this option.

Table 2 shows the constants used in our simulation and analysis along with their source.

Table 2: Constants used in the simulation and analysis.

Name	Value	Unit	Source
Low gas price	0.58	Wei	`https://ethgasstation.info/`, [Online; accessed 21 Jun 2019]
Median gas price	10.5	Wei	`https://ethgasstation.info/`, [Online; accessed 21 Jun 2019]
High gas price	235	Wei	`https://ethgasstation.info/`, [Online; accessed 21 Jun 2019]
Block gas limit	7 997 671	gas	Mean over all (main chain) blocks in 2018

Figures 8, 9, and 10 illustrate our approach to systematically generate the 35 simulation scenarios. Every path from the root to a leaf node represents one of the strategies evaluated. Whenever a node contains a range of batch sizes (BS),

[17] Version 6.4.4; `https://github.com/trufflesuite/ganache-cli`

the actual size is varied in steps of 100 in separate scenarios. Equation 3 documents how the BASE_LINE|INTERNAL_BATCH|PUSH|UNIFORM strategies are computed $G_{baseline}(1000, 100)$ calculates the baseline cost for 1000 recipients and a batch size of 100.

$$
\begin{aligned}
\mathrm{G}_{\mathrm{tx}} &= 21000 \\
\mathrm{G}_{\mathrm{sstorenew}} &= 20000 \\
\mathrm{G}_{\mathrm{zeroinput}} &= 4 \\
\mathrm{G}_{\mathrm{nonzeroinput}} &= 68 \\
\mathrm{n}_{\mathrm{tx}}(n, bs) &= \left\lceil \frac{n}{bs} \right\rceil \\
\mathrm{G}_{\mathrm{inputword}}(b_{set}) &= \mathrm{G}_{\mathrm{nonzeroinput}} \cdot b_{set} + \mathrm{G}_{\mathrm{zeroinput}} \cdot (32 - b_{set}) \\
\mathrm{G}_{\mathrm{inputuniform}}(n) &= (n \cdot \mathrm{G}_{\mathrm{inputword}}(20)) + \mathrm{G}_{\mathrm{inputword}}(2) \\
\mathrm{G}_{\mathrm{sstores}}(n) &= n \cdot \mathrm{G}_{\mathrm{sstorenew}} \\
\mathrm{G}_{\mathrm{txs}}(n, bs) &= \mathrm{n}_{\mathrm{tx}}(n, bs) \cdot \mathrm{G}_{\mathrm{tx}} \\
\mathrm{G}_{\mathrm{baseline}}(n, bs) &= \mathrm{G}_{\mathrm{txs}}(n, bs) + \mathrm{G}_{\mathrm{sstore}}(n) + \mathrm{G}_{\mathrm{inputuniform}}(n)
\end{aligned}
\tag{3}
$$

Equation 1: Calculation of baseline strategy. Only transaction overhead, transaction input and storage writes are considered. Input sizes are hard coded to 20 bytes per address and 2 bytes for the amount.

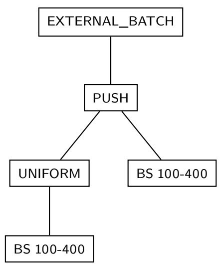

Fig. 8: Externally batched airdrop strategies (8) that were run in the simulation. BS stands for *Batch Size*. Batch size steps are always 100.

The complete simulation code along with analysis scripts and visualization will be released on Github.[18]

[18] https://github.com/soad003/TheOperationalCostOfEthereumAirdrops

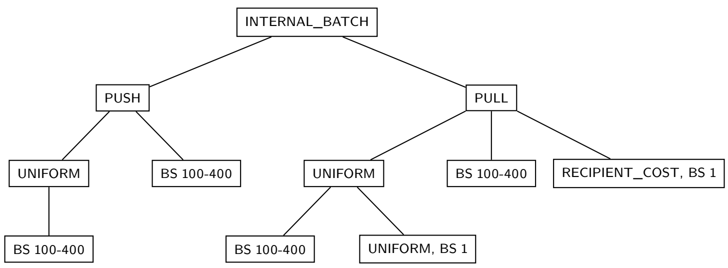

Fig. 9: Internally batched airdrop strategies (18) that were run in the simulation. BS stands for *Batch Size*. Batch size steps are always 100.

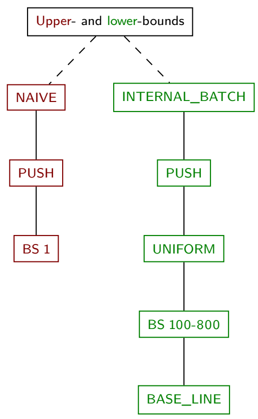

Fig. 10: Airdrop strategies (9) that serve as upper- and lower-bound in the analysis. BS stands for *Batch Size*. Batch size steps are always 100.

B Contracts Used in the Simulation

For our simulation, we use a slightly modified version of the popular OpenZeppelin implementation of an ERC-20 token[19]. We add internal batching to the ERC-20 token by copying the functions `airdrop`, `airdropDynamic`, `airdropApprove`, and `airdropApproveDynamic` from the external batching (`Airdropper.sol`) contract into the ERC-20 token contract. The external batching contract was inspired by a real batching contract.[20] Some additional changes to the original OpenZeppelin implementation where needed in order to make it compile with the current Solidity language (`solc` 5.0.0 and higher). We also changed the visibility of the `mint` function to public in order to be able

[19] https://github.com/OpenZeppelin/openzeppelin-solidity/blob/9b3710465583284b8c4c5d2245749246bb2e0094/contracts/token/ERC20/ERC20.sol, commit: 9b3710465583284b8c4c5d2245749246bb2e0094

[20] https://github.com/iosiro/airdropper/blob/master/contracts/Airdropper.sol, commit: 3667ec866a5310b049c5dcdcd931f046a3203313

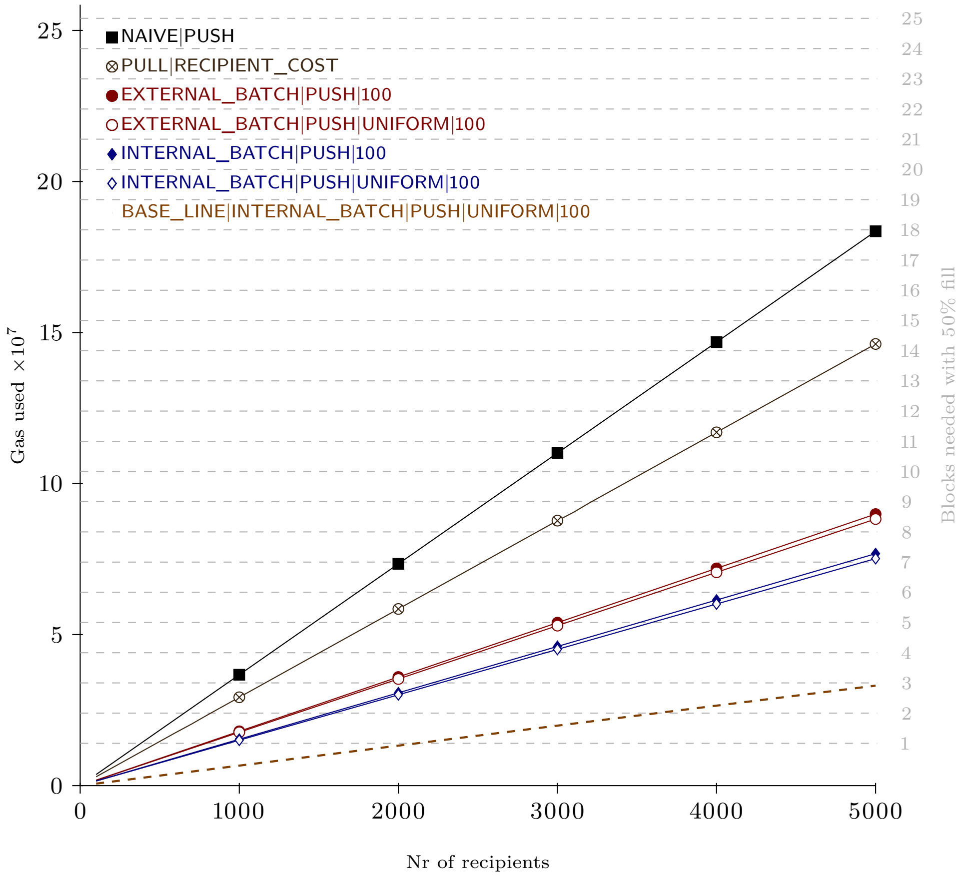

Fig. 11: Discounted version of Figure 5. Discounted pull-style airdrop has been removed from the plot since it would be vulnerable to double spending, see Section 2.3.2.

to generate new tokens when needed in the simulation. The only changes we made to the other source files (`SafeMath.sol`, `IERC20.sol`) were and update of the compiler pragma to version 5.0.0.

All the necessary files including the compiled binaries[21] can be found in the aforementioned Github repository.

[21] Compiled with the Remix IDE, `solc` 0.5.0+commit.1d4f565a with optimizations enabled.