

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КІЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені Ігоря СІКОРСЬКОГО»
Навчально-науковий фізико-технічний інститут
Кафедра математичних методів захисту інформації**

**Звіт до лабораторної №2
за темою:
Реалізація смарт-контракту
та/або анонімної криптовалюти**

**Оформлення звіту:
Юрчук Олексій, ФІ-52МН**

ЗМІСТ

1 Вступ	1
2 Як працюють смарт-контракти (in general)	1
2.1 Визначення	1
2.2 Стан, газ і вартість контрактів (EVM Storage and Gas)	2
2.3 Події (Events)	2
2.4 Діаграма життєвого циклу смарт-контракту	2
3 Ballot Contract	2
3.1 Призначення договору	2
3.2 Структури даних	2
3.2.1 <code>struct Proposal</code> (пропозиція)	3
3.2.2 <code>struct Voter</code>	3
3.3 Станові змінні	3
3.4 Конструктор	4
3.5 Функція: <code>giveRightToVote(address voter)</code>	4
3.6 Функція: <code>delegate(address to)</code>	4
3.7 Функція: <code>vote(uint proposal)</code>	5
3.8 Функції перегляду: <code>winningProposal()</code> and <code>winnerName()</code>	6
3.9 Діаграма взаємодії контрактів	6
4 Оптимізація газу	7
4.1 Оптимізований контракт: <code>BallotOptimized.sol</code>	7
4.2 Підсумки оптимізації	10
4.3 Порівняння газових витрат	11

1 Вступ

За мету в цій лабораторній роботі я ставив собі попрацювати і набути практичних навичок зі смарт-контрактами Ethereum (у продовження першої лаби). У Завданні 1 я візьму один з еталонних контрактів з офіційної документації Solidity (версія 0.5.3, «Solidity by Example»), розгорну його в тестовій мережі Sepolia та спробую покращити його газову ефективність. А потім ще спробую накопичити ще контракт власними силами (а-ля завдання 2).

На [сайті Solidity](#) наведено такі приклади контрактів:

1. **Voting (Ballot)** — система делегованого голосування з контролем доступу.
2. **Simple Auction** — відкритий аукціон з ефірним депозитом.
3. **Blind Auction** — аукціон з розкриттям ставок після їхнього підтвердження з хешованими ставками.
4. **Payment Channel** — позаланцюгові мікроплатежі з підписами.

Зупиняється я на контракті **Ballot (Voting)** (Голосування), оскільки він демонструє найширший спектр можливостей мови Solidity – структури, відображення, контроль доступу, логіку делегування та функції перегляду, і при цьому не вимагає передачі Ether, що спрощує тестування. Крім того, його механізм делегування містить чітку можливість оптимізації газу (необмежений цикл `while`).

2 Як працюють смарт-контракти (in general)

2.1 Визначення

Smart contract це певна програма, що зберігається в блокчейні Ethereum і виконується автоматично при виклику її функцій. Життєвий цикл складається з чотирьох етапів:

1. **Writing.** Розробник пише вихідний код у Solidity (`.sol`). Компілятор (`solc`) створює два артефакти:
 - *EVM bytecode* — низькорівневі інструкції, що виконуються віртуальною машиною Ethereum.
 - *ABI (Application Binary Interface)* — опис у форматі JSON усіх публічних функцій, їхніх параметрів та типів повернення. Зовнішні інструменти, такі як (web3.js, ethers.js, Remix), використовують ABI для кодування/декодування викликів.
2. **Deployment.** Спеціальна транзакція з порожнім полем `to` надсилає байт-код до мережі. EVM виконує `constructor()` рівно один раз, ініціалізує змінні стану, а мережа присвоює контракту постійну адресу.
3. **Interaction.** Користувачі (або інші контракти) викликають функції одним із двох способів:
 - *State-changing functions (write)* — створюють транзакцію, яка має бути підтверджена в блокчейні; (cost gas).
 - *View / pure functions (read)* — виконується локально by node, (free of charge).
4. **Termination.** Контракт існує в ланцюжку безстроково. Його можна зробити нефункціональним за допомогою операційного коду `selfdestruct` (застарілого в сучасній Solidity) або шляхом впровадження шаблону паузи/знищення.

2.2 Стан, газ і вартість контрактів (EVM Storage and Gas)

Стан контракту зберігається в постійній пам'яті (*storage slots*), кожен з яких має ширину 256 біт. Запис у новий слот коштує 20000 gas (`SSTORE`); оновлення існуючого слота коштуватиме 5 000 газу; читання коштує 2 100 газу (`SLOAD`). Пам'ять (тимчасова, для кожного виклику) і дані виклику (тільки для читання, для аргументів функції) значно дешевші.

Кожен виконуваний EVM код має фіксовану газову вартість. Відправник транзакції вказує *gas limit*, а також сплачує $\text{gas used} \times \text{gas price}$ в ETH. Якщо газ закінчується в процесі виконання, всі зміни стану скасовуються, і (що важливо!) спожитий газ на рахунок *не* повертається. Цей механізм запобігає нескінченним циклам і стимулює писати ефективний код.

2.3 Події (Events)

Контракти можуть `emit` (продукувати) певні події. Ці події зберігаються в ланцюжку як журнали транзакцій (це набагато дешевше, ніж зберігання: $\sim 375 \text{ gas base} + 375 \text{ gas per indexed topic} + 8 \text{ gas per byte of data}$). Вони доступні для запиту поза ланцюжком, але *не* доступні для читання іншими контрактами.

2.4 Діаграма життєвого циклу смарт-контракту

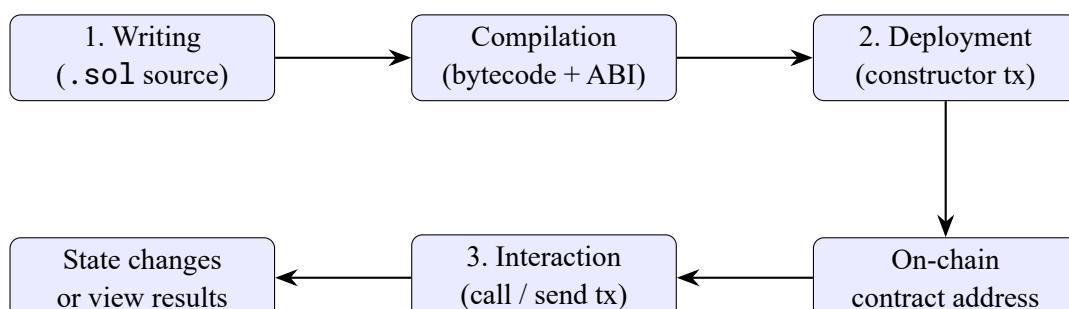


Рис. 1: Smart contract lifecycle on Ethereum.

3 Ballot Contract

Даний розділ я хочу присвятити детальному огляну оригінального контракту `Ballot` з офіційної документації Solidity 0.5.3, включаючи його структури даних, змінні стану та функції.

3.1 Призначення договору

Контракт `Ballot` реалізує *delegated voting* (делеговане голосування). Головуючий створює бюллетень із переліком пропозицій. Він надає право голосу окремим адресам. Кожен уповноважений виборець може або проголосувати безпосередньо за пропозицію, або делегувати (передати) свій голос іншому виборцю, якому він довіряє. В кінці будь-хто може дізнатися, яка пропозиція перемогла.

3.2 Структури даних

3.2.1 struct Proposal (пропозиція)

```
1 struct Proposal {  
2     bytes32 name;      // Short name (up to 32 bytes)  
3     uint voteCount;   // Accumulated votes  
4 }
```

Тут використовується тип `bytes32` замість `string`, оскільки він займає рівно один 256-бітний слот пам'яті, тоді як `string` вимагає динамічного розподілу пам'яті і є значно дорожчим у газовому вираженні.

3.2.2 struct Voter

```
1 struct Voter {  
2     uint weight;       // Voting power (0 = no right, 1+ = can vote)  
3     bool voted;        // True after voting or delegating  
4     address delegate; // Delegation target; address(0) if none  
5     uint vote;          // Index of chosen proposal (meaningful only  
6                           // if voted==true && delegate==address(0))  
7 }
```

Поле	Опис
<code>weight</code>	Право голосу. Ініціалізується 0 для всіх адрес. Встановлюється 1 головуючим за допомогою функції <code>giveRightToVote()</code> . Збільшується, коли інші виборці делегують право голосу цій адресі.
<code>voted</code>	Boolean flag. Встановлюється на <code>true</code> як у випадку, коли виборець голосує сам <i>або</i> коли він делегує свої повноваження. Запобігає подвійному голосуванню.
<code>delegate</code>	Адреса Ethereum, який виборець делегував свої повноваження. Залишається 0, якщо виборець проголосував самостійно або ще не вчинив жодних дій
<code>vote</code>	Index в масиві <code>proposals[]</code> . Використовується лише тоді, коли виборець голосує безпосередньо (не делегує).

3.3 Станові змінні

```
1 address public chairperson;  
2 mapping(address => Voter) public voters;  
3 Proposal[] public proposals;
```

Змінна	Тип	Призначення
<code>chairperson</code>	<code>address</code>	Зберігає адресу розробника контракту. Тільки ця адреса може надавати право голосу.
<code>voters</code>	<code>mapping</code>	Відображає кожну адресу Ethereum у структуру <code>Voter</code> .
<code>proposals</code>	<code>Proposal[]</code>	Динамічний масив всіх пропозицій. Створюється конструктором один раз.

Примітка. Відображення не зберігає ключі, тому неможливо пройти по всіх виборцях в ланцюжку.

3.4 Конструктор

```
1 constructor(bytes32[ ] memory proposalNames) public {
2     chairperson = msg.sender;
3     voters[chairperson].weight = 1;
4
5     for (uint i = 0; i < proposalNames.length; i++) {
6         proposals.push(Proposal({
7             name: proposalNames[i],
8             voteCount: 0
9         }));
10    }
11 }
```

Конструктор виконується *exactly once* під час розгортання контракту. Він записує `msg.sender` як головуючого, автоматично надає йому вагу голосу 1 і заповнює масив `proposals` з наданих йому імен. Вартість газу для розгортання пропорційна кількості пропозицій (кожен `push` записує новий слот пам'яті).

3.5 Функція: `giveRightToVote(address voter)`

```
1 function giveRightToVote(address voter) public {
2     require(msg.sender == chairperson,
3            "Only chairperson can give right to vote.");
4     require(!voters[voter].voted,
5            "The voter already voted.");
6     require(voters[voter].weight == 0);
7     voters[voter].weight = 1;
8 }
```

Функція застосовує три заходи безпеки:

1. Тільки головуючий може її викликати (access control).
2. Цільова адреса must not already possess voting rights (я хз як це перекласти нормально).
3. Ціль не повинна ще мати права голосу (`weight == 0`).

У разі успіху вага виборця встановлюється на 1, що дозволяє йому брати участь у голосування. Приблизна вартість газу складає: $\sim 47\,000$ (запис нового слота пам'яті).

Обмеження: Кожен виборець потребує окремої транзакції. Для N виборців головуючий сплачує $N \times (\sim 47\,000 + 21\,000_{\text{base}})$ газу. У самій документації Solidity розробники ставлять нам питання: "Чи можете ви придумати кращий спосіб?" – на це питання я дам в розділі з оптимізацією.

3.6 Функція: `delegate(address to)`

Це, певно, найскладніша функція в контракті. Виглядає вона наступним чином:

```

1   function delegate(address to) public {
2       Voter storage sender = voters[msg.sender];
3       require(!sender.voted, "You already voted.");
4       require(to != msg.sender, "Self-delegation is disallowed.");
5
6       while (voters[to].delegate != address(0)) {
7           to = voters[to].delegate;
8           require(to != msg.sender, "Found loop in delegation.");
9       }
10
11      sender.voted = true;
12      sender.delegate = to;
13      Voter storage delegate_ = voters[to];
14
15      if (delegate_.voted) {
16          proposals[delegate_.vote].voteCount += sender.weight;
17      } else {
18          delegate_.weight += sender.weight;
19      }
20  }

```

1. Завантажує структуру `Voter` виборця за посиланням (storage pointer).
2. Перевіряє, що виборець ще не голосував і не делегує сам собі.
3. **Розв'язує ланцюжок делегування:** Цикл `while` проходить ланцюг делегування ($A \rightarrow B \rightarrow C$) до тих пір, поки не знайде виборця, який не делегує далі. Цей цикл також перевіряє наявність циклічного делегування.
4. Позначає виборця як проголосованого та записує остаточного делегата.
5. **Два результати:**
 - Якщо остаточний делегат вже голосував, вага виборця додається безпосередньо до обраної ним пропозиції.
 - Якщо остаточний делегат ще не голосував, вага виборця переноситься до делегата (накопичуючи voting power).

газові ризики: Цикл `while` зчитує storage, сплачуєчи ($SLOAD = 2\,100\ gas$) на кожній ітерації. Ланцюжок делегування довжиною k коштує приблизно $k \times 2\,100\ gas$ лише за один цикл. Чим довше ланцюжок, тим більша ймовірність вичерпати газовий ліміт блоку.

3.7 Функція: `vote(uint proposal)`

```

1   function vote(uint proposal) public {
2       Voter storage sender = voters[msg.sender];
3       require(sender.weight != 0, "Has no right to vote");
4       require(!sender.voted, "Already voted.");
5
6       sender.voted = true;
7       sender.vote = proposal;

```

```

8     proposals[proposal].voteCount += sender.weight;
9 }
```

Все доволі просто: позначити виборця як проголосувавшого, записати його вибір і додати його вагу до підрахунку обраної пропозиції. Якщо індекс `proposal` виходить за межі масиву, Solidity автоматично викликає `revert`, скасовуючи всі зміни стану. Вартість по газу: ~47 000–65 000 залежно від того, чи слоти пам'яті є новими або вже записаними.

3.8 Функції перегляду: `winningProposal()` and `winnerName()`

```

1   function winningProposal() public view
2       returns (uint winningProposal_)
3   {
4       uint winningVoteCount = 0;
5       for (uint p = 0; p < proposals.length; p++) {
6           if (proposals[p].voteCount > winningVoteCount) {
7               winningVoteCount = proposals[p].voteCount;
8               winningProposal_ = p;
9           }
10      }
11  }
12
13 function winnerName() public view
14     returns (bytes32 winnerName_)
15 {
16     winnerName_ = proposals[winningProposal()].name;
17 }
```

Обидві функції є типу `view`. Це означає, що вони не змінюють стан і *free to call* (без транзакції, без витрати газу) ззовні. Функція `winningProposal()` ітерує по всіх пропозиціях та повертає індекс тієї, яка має найбільшу кількість голосів (`voteCount`). Функція `winnerName()` викликає вищезгадану для отримання індексу переможця, а потім просто повертає його ім'я.

Обмеження: Не враховує рівність голосів! Якщо є випадок, де дві пропозиції мають однакову кількість голосів, перемогу отримує та, яка має нижчий індекс.

3.9 Діаграма взаємодії контрактів

Візуалізувати це можна наступним чином:

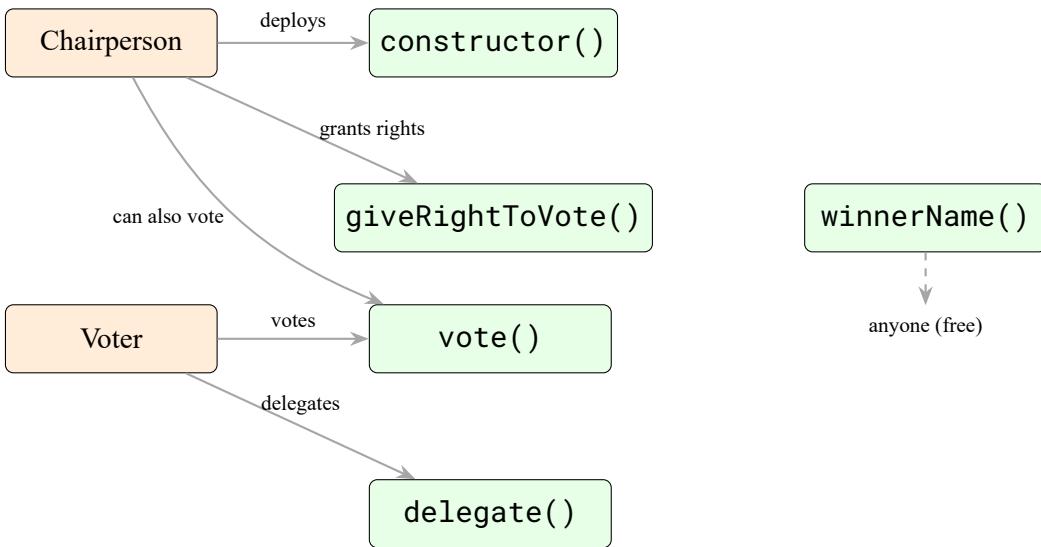


Рис. 2: Interaction roles in the Ballot contract.

4 Оптимізація газу

Оригінальний контракт Ballot, з навчальної точки зору ж достатньо зрозумілим, проте все ж містить кілька можливостей для скорочення витрат газу.

4.1 Оптимізований контракт: `BallotOptimized.sol`

```

1 // SPDX-License-Identifier: GPL-3.0
2 pragma solidity >=0.7.0 <0.9.0;
3
4 /// @title Optimized Voting with delegation
5 contract BallotOptimized {
6
7     struct Voter {
8         uint weight;
9         bool voted;
10        address delegate;
11        uint vote;
12    }
13
14    struct Proposal {
15        bytes32 name;
16        uint voteCount;
17    }
18
19    address public chairperson;
20    mapping(address => Voter) public voters;
21    Proposal[] public proposals;
22
23    // OPT-1: Events for off-chain tracking (cheaper than storage)
24    event VoterAuthorized(address voter);
25    event VoteCast(address voter, uint proposal);

```

```

26     event VoteDelegated(address from, address to);
27
28     constructor(bytes32[] memory proposalNames) {
29         chairperson = msg.sender;
30         voters[chairperson].weight = 1;
31
32         for (uint i = 0; i < proposalNames.length; i++) {
33             proposals.push(Proposal({
34                 name: proposalNames[i],
35                 voteCount: 0
36             }));
37         }
38     }
39
40     // OPT-2: external instead of public
41     function giveRightToVote(address voter) external {
42         require(
43             msg.sender == chairperson,
44             "Only chairperson can give right to vote."
45         );
46         require(!voters[voter].voted, "The voter already voted.");
47         require(voters[voter].weight == 0);
48         voters[voter].weight = 1;
49         emit VoterAuthorized(voter);
50     }
51
52     // OPT-3: Batch granting of rights
53     function giveRightToVoteBatch(address[] calldata voterList)
54         external
55     {
56         require(msg.sender == chairperson, "Only chairperson.");
57         for (uint i = 0; i < voterList.length; i++) {
58             address v = voterList[i];
59             if (!voters[v].voted && voters[v].weight == 0) {
60                 voters[v].weight = 1;
61                 emit VoterAuthorized(v);
62             }
63         }
64     }
65
66     function delegate(address to) external {
67         Voter storage sender = voters[msg.sender];
68         require(!sender.voted, "You already voted.");
69         require(to != msg.sender, "Self-delegation is disallowed.");
70
71         // OPT-4: Limit chain depth to prevent gas blowup
72         uint chainLength = 0;
73         address current = to;
74         while (voters[current].delegate != address(0)) {
75             current = voters[current].delegate;
76             require(

```

```

77         current != msg.sender,
78         "Found loop in delegation."
79     );
80     chainLength++;
81     require(chainLength <= 10, "Delegation chain too long");
82 }
83
84 sender.voted = true;
85 // OPT-5: Point directly to chain end
86 sender.delegate = current;
87 Voter storage delegate_ = voters[current];
88
89 if (delegate_.voted) {
90     // OPT-6: unchecked arithmetic
91     unchecked {
92         proposals[delegate_.vote].voteCount += sender.weight;
93     }
94 } else {
95     unchecked {
96         delegate_.weight += sender.weight;
97     }
98 }
99 emit VoteDelegated(msg.sender, current);
100}
101
102function vote(uint proposal) external {
103    Voter storage sender = voters[msg.sender];
104    require(sender.weight != 0, "Has no right to vote");
105    require(!sender.voted, "Already voted.");
106
107    sender.voted = true;
108    sender.vote = proposal;
109
110    unchecked {
111        proposals[proposal].voteCount += sender.weight;
112    }
113    emit VoteCast(msg.sender, proposal);
114}
115
116function winningProposal() public view
117    returns (uint winningProposal_)
118{
119    uint winningVoteCount = 0;
120    // OPT-7: Cache array length in memory
121    uint len = proposals.length;
122    for (uint p = 0; p < len;) {
123        if (proposals[p].voteCount > winningVoteCount) {
124            winningVoteCount = proposals[p].voteCount;
125            winningProposal_ = p;
126        }
127        // OPT-8: Unchecked loop increment

```

```

128         unchecked { p++; }
129     }
130 }
131
132 function winnerName() external view
133     returns (bytes32 winnerName_)
134 {
135     winnerName_ = proposals[winningProposal()].name;
136 }
137

```

4.2 Підсумки оптимізації

№	Оптимізація	Заощадження (газу)	Пояснення
1	Події замість додаткового зберігання	~15 000/подія	Логи коштують ~375 + $375 \times \text{topics}$ + 8/байт; в т.ч. як SSTORE коштує 20 000 для нового слота.
2	external замість public	~200–600/виклик	Зовнішні функції читають аргументи з calldata напряму; в т.ч. як публічні копіюють їх в memory , що дорожче.
3	Batch giveRightToVote	~21 000/виборець	Базова вартість однієї транзакції (21 000 gas) розділена між N виборцями замість N окремих транзакцій.
4	Ліміт глибини ланцюга делегування (10)	Запобігає DoS атакам	Вихідний контракт міг вичерпати ліміт газу блоку на зловмисно довгих ланцюгах.
5	Пряме делегування до кінця ланцюга	Зберігає майбутні проходження	Якщо A → B → C, то зберігає A.delegate = C напряму.
6	unchecked арифметика	~30–60/операція	Вага та кількість голосів не можуть переповнити uint256 в реальних умовах; skip перевірок економить газ.
7	Кешування proposals.length	~100/ітерація	Уникає повторного SLOAD для слота довжини в масив.
8	Безконтрольне збільшення циклу	~30/ітерація	Індекс циклу не може переповнитися в реальних умовах, тому можна пропустити перевірку на переповнення.

4.3 Порівняння газових витрат

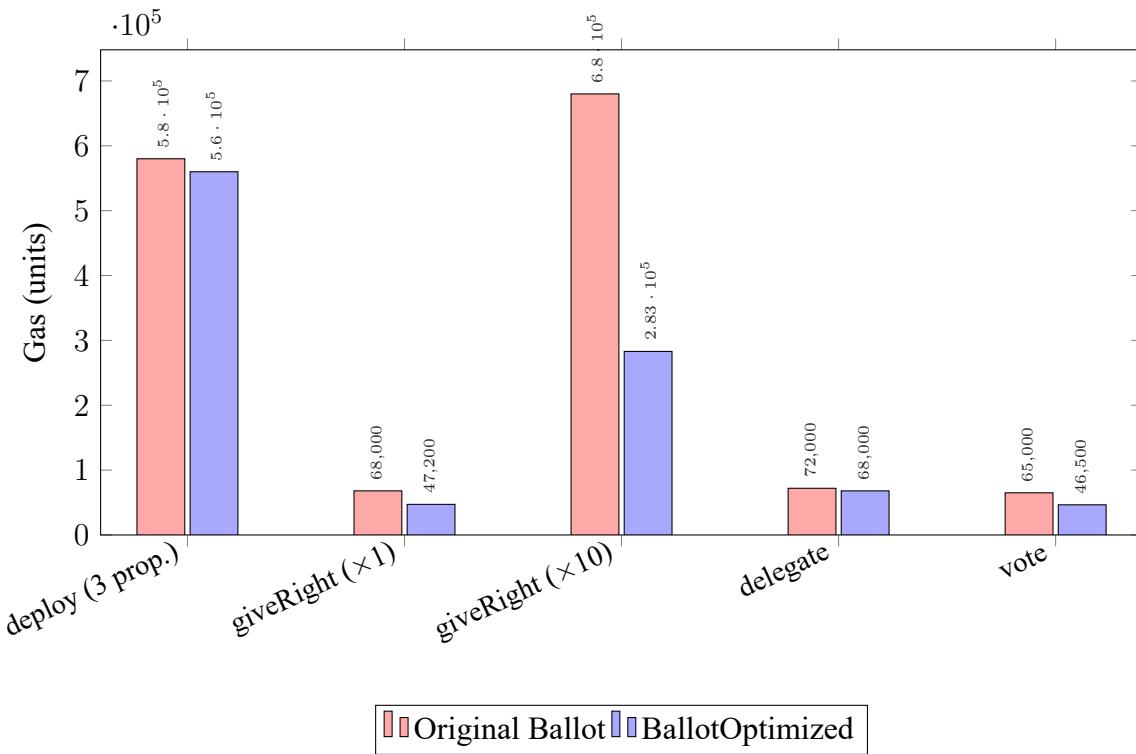


Рис. 3: Approximate gas costs: original vs. optimised Ballot contract.

Найбільш істотна економія гасу вже відчувається при наданні прав голосування 10 виборцям в рамках однієї пакетної транзакції.