



UNIVERSITY OF PADOVA

DEPARTMENT OF MATHEMATICS "TULLIO LEVI-CIVITA"

MASTER THESIS IN CYBERSECURITY

ATTACKING ANONYMITY SET IN TORNAO CASH VIA WALLET FINGERPRINTS

SUPERVISOR

MAURCO CONTI
UNIVERSITY OF PADOVA

CO-SUPERVISOR

ANKIT GANGWAL

MASTER CANDIDATE

MARTINA SOLETI

STUDENT ID

2063554

ACADEMIC YEAR

2023-2024

“GOD’S SOLDIER”

Abstract

Tornado Cash is a decentralized application (dApp) that runs on *Ethereum Virtual Machine* (EVM) compatible networks to enhance users' privacy in terms of user transaction history over the blockchain. The dApp achieves this goal by enabling users to deposit currencies into designated pools and subsequently withdraw them, severing the link between depositor and withdrawer addresses. At deposit time, Tornado Cash communicates to users the level of privacy they will benefit from (*anonymity set*) by depositing currencies into one of its pools. Existing analyses have indicated discrepancies between the claimed *anonymity set* and the actual level of privacy provided, primarily attributed to users' incorrect utilization of the dApp. The current project aims to explore a new way to challenge the dApp proposed *anonymity set* by examining wallet fingerprints, a factor not directly related to user behavior within the application. The findings of this research shed light on the potential for creating links between clusters of users in TC according to the new proposed approach and raise a privacy concern within the Ethereum network.

Contents

ABSTRACT	v
LIST OF FIGURES	ix
LIST OF TABLES	xiii
LISTING OF ACRONYMS	xv
1 INTRODUCTION	1
2 BACKGROUND	5
2.1 Ethereum	6
2.2 Gas fees	8
2.3 Tornado Cash	11
2.3.1 Deposit	12
2.3.2 Withdraw	15
2.4 Useful definitions	19
3 RELATED WORKS	21
4 PROJECT CORE	27
4.1 Wallet analysis	28
4.1.1 Metamask	28
4.1.2 Trust Wallet	33
4.1.3 ShapeShift Wallet	36
4.1.4 Rainbow	37
4.1.5 OneKey	40
4.1.6 Unstoppable Wallet	42
4.2 Tornado Cash analysis	45
4.3 Additional work	52
5 RESULTS	57
6 CONCLUSION	69
ACKNOWLEDGMENTS	73

Listing of figures

2.1	Legacy transaction on Etherscan.	10
2.2	EIP-1559 transaction on Etherscan.	11
2.3	Example of the Tornado Cash 1 ETH pool: addresses A through F deposit to and withdraw from the pool. It quickly becomes impossible to associate withdraw and deposit transactions given a growing mixer [1].	12
2.4	TC pools for ETH currency in the Ethereum mainnet network.	13
2.5	TC smart contracts addresses for ETH currency in the Ethereum mainnet network [2].	13
2.6	Example of TC secret note, received after a deposit in the Goerli testnet network.	14
2.7	Toy example of a TC Merkle tree with height 3, four deposits C_i and four pre-initialized zeros leaf x_i	14
2.8	Toy example of a TC deposit in the Merkle tree (C_5), with related Merkle path $\{x_6, x'_4, x''_1\}$	15
2.9	TC general working schema [3].	18
2.10	Example of a withdrawal from 0.1 ETH pool involving a relayer.	18
3.1	Heuristic 1 schema: a single address A withdrawing and depositing to the same TC pool [1].	23
3.2	Heuristic 2 schema: two addresses (A and D) depositing to and withdrawing from the same TC pool with an equal custom-set gas price [1].	23
3.3	Heuristic 3 schema: addresses A and D deposit and withdraw from the same TC pool, moreover interactions in terms of transactions do exist between them out of TC [1].	24
3.4	Heuristic 4 schema: addresses A and D deposit and withdraw the same number of times from the same three Tornado Cash pools [1].	24
3.5	Heuristic 5 schema: address D performs a withdrawal from the 1 ETH pool, obtaining a reward implying that the deposit has been in the pool for n blocks. The only deposit present in the prior n block is the one made by A , so the two addresses are linked [1].	25
4.1	<i>Metamask</i> features at frontend level.	29
4.2	<i>Metamask</i> entry point for the <i>send</i> transaction button.	30
4.3	<i>Metamask</i> 's code flow in case of gas fee suggestions for type-2 transactions.	30
4.4	<i>Metamask</i> 's code flow in case of gas fee suggestions for type-0 transactions.	30

4.5	Figure 4.3's <i>catch</i> branch ending flow, repeated for each priority level.	32
4.6	Figure 4.5 additional.	32
4.7	<i>Metamask</i> frontend gas fee suggestions for the low priority level along a type-2 transaction.	33
4.8	Collection of <i>Metamask</i> gas fee suggestions from the two URLs inherent to a type-2 transaction.	33
4.9	<i>Trust Wallet</i> 's RPC parameters.	34
4.10	<i>Trust Wallet</i> 's RPC parameters.	35
4.11	<i>Trust Wallet</i> frontend gas fee suggestions for a type-2 transaction.	35
4.12	Collection of <i>Trust Wallet</i> gas fee suggestion for a type-2 transaction through a <i>python</i> script emulating the logic of <i>Trust Wallet</i>	35
4.13	Example of results coming from the <i>ShapeShift Wallet</i> API.	36
4.14	<i>ShapeShift Wallet</i> gas fee suggestions at the frontend level along a transfer operation.	37
4.15	<i>ShapeShift Wallet</i> gas fee suggestions retrieved through a <i>Python</i> script.	37
4.16	<i>Rainbow</i> 's source code snippet with the reference to the API involved in the gas fee suggestions.	38
4.17	Output form of the Figure 4.16's API call.	38
4.18	Priority levels multipliers.	39
4.19	<i>Rainbow</i> wallet suggestions for the <i>normal</i> level.	40
4.20	<i>Rainbow</i> wallet suggestions for the <i>fast</i> level.	40
4.21	<i>Rainbow</i> wallet suggestions for the <i>urgent</i> level.	40
4.22	<i>Rainbow</i> gas fee predictions retrieved through a Python script. For the <i>maxFeePerGas</i> parameter, the picture shows both the float value and its rounding to the nearest integer.	40
4.23	<i>OneKey</i> wallet suggestions for the <i>low</i> level.	41
4.24	<i>OneKey</i> wallet suggestions for the <i>normal</i> level.	41
4.25	<i>OneKey</i> wallet suggestions for the <i>high</i> level.	41
4.26	Blocknative gas fee predictions for ranges of confidence in descending order (from 90% to 70%).	42
4.27	<i>Unstoppable Wallet</i> RPC parameters.	43
4.28	Figure 4.27's results handling.	43
4.29	Definition of <i>recommendedBaseFee</i> function in Figure 4.28.	44
4.30	Definition of <i>recommendedPriorityFee</i> function in Figure 4.28.	44
4.31	<i>Unstoppable Wallet</i> gas fee suggestions at frontend level.	45
4.32	<i>Unstoppable Wallet</i> gas fee suggestions retrieved through a python script.	45
4.33	Portion of code related to the Tornado Cash's <i>generateTransaction</i> function.	46
4.34	Portion of code related to the Tornado Cash's <i>fetchGasPrice</i> function.	47
4.35	Portion of code related to the Tornado Cash's <i>gasPriceETH</i> function.	47
4.36	Code snippet 1 of the <i>gasPrice</i> function in <i>gas-price-oracle</i> external library.	48

4.37	Oracle 1 taken into account by snippet in Figure 4.36.	48
4.38	Oracle 2 taken into account by snippet in Figure 4.36.	48
4.39	Code snippet 2 of the <i>gasPrice</i> function in <i>gas-price-oracle</i> external library. . .	49
4.40	Oracle taken into account by snippet in Figure 4.39.	49
4.41	Multipliers for different priority levels generation valid for both Figure 4.39 and Figure 4.42 snippets.	50
4.42	Code snippet 3 of the <i>gasPrice</i> function in <i>gas-price-oracle</i> external library. . .	50
4.43	Example of a withdrawal involving a relayer and the 100 ETH smart contract. . .	51
4.44	Tornado Cash gas fee suggestions according to the above formula.	51
4.45	Example of TC activity concerning its 10 ETH pool in <i>Ethereum Mainnet</i> network.	52
4.46	Example of TC activity concerning its 100 MATIC pool in <i>Polygon</i> network. . .	53
4.47	<i>Rabby Wallet</i> suggested fees at frontend level.	55
4.48	<i>Rabby Wallet</i> suggested fees in <i>wei</i> retrieved through the API.	55
5.1	Example of gas fee suggestions collected at a specific UTC time by the built <i>Python</i> script.	58
5.2	<i>Javascript</i> code snippet for retrieving (and filter) transactions from the <i>Ethereum</i> blockchain.	60
5.3	Example of a real transaction validated over the <i>Ethereum</i> Blockchain.	61
5.4	Gas fee suggestion collections coming from the analyzed wallets for a times- tamp consistent with the transaction in Figure 5.3.	61
5.5	Example of a real transaction validated over the <i>Ethereum</i> Blockchain.	61
5.6	Gas fee suggestion collections coming from the analyzed wallets for a times- tamp consistent with the transaction in Figure 5.5.	61
5.7	Example of a real transaction validated over the <i>Ethereum</i> Blockchain.	62
5.8	Gas fee suggestion collections coming from the analyzed wallets for a times- tamp consistent with the transaction in Figure 5.7.	62
5.9	Example of a real deposit transaction validated over the <i>Ethereum</i> Blockchain. TC pool involved is 100 ETH.	65
5.10	Full match in terms of gas fee suggestions coming from the <i>Metamask</i> wallet for a timestamp consistent with the transaction in Figure 5.9.	65
5.11	Example of a real withdrawal transaction validated over the <i>Ethereum</i> Blockchain. TC pool involved is 100 ETH.	65
5.12	Full match in terms of gas fee suggestions coming from the <i>Metamask</i> wallet for a timestamp consistent with the transaction in Figure 5.11.	65
5.13	Example of a real Tornado Cash deposit transaction validated over the <i>Ethereum</i> Blockchain.	67
5.14	Gas fee suggestion collections coming from the TC platform according to the empirically retrieved formula for a timestamp consistent with the transaction in Figure 5.13.	67

5.15	Example of a real Tornado Cash withdrawal transaction with relayer involved validated over the <i>Ethereum</i> Blockchain.	67
5.16	Gas fee suggestion collections coming from the TC platform according to the empirically retrieved formula for a timestamp consistent with the transaction in Figure 5.15.	67
5.17	Example of a real Tornado Cash withdrawal transaction with no relayer involved validated over the <i>Ethereum</i> Blockchain.	68
5.18	Gas fee suggestion collections coming from the TC platform according to the empirically retrieved formula for a timestamp consistent with the transaction in Figure 5.17.	68

Listing of tables

Listing of acronyms

dApp	Decentralized Application
TC	Tornado Cash
DeFi	Decentralized Finance
P2P	Peer to Peer
EOA	Externally Owned Account
ENS	Ethereum Name Service
EVM	Ethereum Virtual Machine
OFAC	Office of Foreign Assets Control
SDN	Specially Designated Nationals And Blocked Persons
PoW	Proof-of-Work
zk-SNARK	Zero-Knowledge Succint Non Interactive Argument of Knowledge
RPC	Remote Procedure Call
CLI	Command Line Interface
EIP	Ethereum Improvement Proposal
GNN	Graph Neural Network

1

Introduction

Blockchains are public, decentralized, distributed, append-only immutable ledgers that provide users with pseudo-anonymity, enabling them to trigger some events that are recorded in the form of a new transaction written on the ledger. Money transfer is an event example over the blockchain. The term *blockchain* itself refers to the ledger structure: a chain of blocks, with each block containing transactions and other data. This chain is made in such a way that anyone can get in it and look at its content (public), there is not a single entity that detains control over it (decentralized), anyone can have a copy of it according to a peer-to-peer protocol (distributed) and new content can take part of it considering that, once added, no modification is allowed (append-only). Users are drawn to blockchains for their decentralized nature, freeing assets from centralized authorities like banks, and for the pseudo-anonymity they offer, with each user identified by one or more hexadecimal addresses. Bitcoin [4] is the first blockchain system to go live, enabling parties to engage in money transfers using the native currency of the blockchain. Bitcoin has been followed up by Ethereum, a blockchain offering enhanced capabilities by enabling the execution of decentralized applications (dApps) directly within its blockchain network through the deployment of smart contracts [5]. Smart contracts consist of programs runnable over the blockchain, there identified by their unique address. An example of smart contract running over Ethereum is Tornado Cash (TC). It consists of a dApp that retrofits the network with privacy, addressing concerns arising from the pseudo-anonymity (and not anonymity) offered by blockchains: a malicious user could analyze the blockchain public data for inferring correlations between addresses or even the identity of users behind

some addresses. This result could be achieved with the help of off-chain data, enabling the malicious user to profile other blockchain users and to understand who owns what [2]. Tornado Cash is part of the family of privacy mixers, solutions born to make funds untraceable. If on one hand such a solution is legitimately embraced by users who are willing to increase their privacy over the blockchain, on the other hand the provided untraceability property has led to the abuse of mixing services for money laundering and committing fraud. These illegal actions, which pose significant threats to the blockchain ecosystem and financial order [3], have captured the interest of centralized regulator, leading on August 8th 2022 the US Treasury's *Office of Foreign Assets Control* (OFAC) to place sanctions over TC due to alleged facilitation of money laundering. OFAC added the TC website and related blockchain addresses to the *Specially Designated Nationals And Blocked Persons* (SDN) list. According to the sanctions, US citizens are no longer legally allowed to use the TC website or involve any property or interest transactions with those blacklisted addresses [6]. The sanctions have led to a series of consequences, like miners who stopped processing any TC deposit and withdrawal transactions, as well as *decentralized finance application* (DeFi) platforms which started banning addresses that receive transactions from TC. Such a censorship has greatly reduced the mixer's daily deposits but has not completely stopped them [7], implying that they do exist methods to bypass it. At the frontend level, for instance, DeFi users could interact with the platform smart contracts through a *Command Line Interface* (CLI) or could fork the platform project to create their own frontend interface [8]. Bypassing the censorship makes the problem of the illegal activities in which mixers could be involved still actual, this is one of the reasons that led researchers to put their attention into ways to reconstruct the linkability broken by privacy mixers, hence creating back a correlation between mixer's users to regain funds traceability. For this purpose, several heuristics have already been proposed (Chapter 3 for details). The proposed heuristics lay down on the way TC mixer is approached by users, hence behavioral errors made by users that make the dApp not express itself at its best. The aim of the current project is that of going on with the open research in this field, moving the focus to a novel approach that takes into account fingerprints left by wallets at transaction time to create linkable clusters of users: those users who have made a transaction through the same wallet software are considered to belong to the same cluster, hence to be linkable. Translating this into the TC context and moving under the assumption that when a user makes use of TC he will use the same wallet software for both his deposit and withdrawal, deposits and withdrawals starting from the same wallet software are considered to be linkable.

The contributions of this thesis are threefold:

- A new Ethereum privacy concern is identified and validated through empirical testing;
- A new heuristic to create linkable clusters of TC users based on wallet fingerprints has been built up. This new approach is distinct from the existing ones since it is not directly related to the user approaching the dApp;
- Existing TC transactions (deposits and withdrawals) covering a time window of about one month have been analyzed to evaluate the effectiveness of the proposed heuristic.

To provide a comprehensive understanding of the research journey and its contributions, the remainder of this thesis is structured as follows: Chapter 2 provides the reader with a strong background on the Ethereum network, elucidating its working and the role of TC within it. This chapter delves into technical details relevant to the context. Chapter 3 offers a summary of existing research efforts aimed at reconstructing address linkability disrupted by mixers like TC. It underscores that previous heuristics primarily focus on how users engage with the TC dApp. Chapter 4 presents the concrete efforts undertaken in this thesis project. It outlines the development of a novel approach targeting TC's anonymity set by leveraging wallet fingerprints. This chapter details the methodology employed and the rationale behind the proposed approach. Chapter 5 evaluates the proposed methodology by analyzing its impact on both the Ethereum network and TC transactions. This evaluation aims to investigate the efficacy and feasibility of the approach, showcasing the results obtained. Chapter 6 serves as the culmination of this work, summarizing the content and contributions of the thesis. Additionally, it offers insights into potential future directions for research in this domain.

2

Background

Technical details about different blockchains may vary, but intuitively a blockchain is a special type of database that is shared between nodes in a peer-to-peer network, where a node can be represented by any user who owns a device running a specific client software to take part to the blockchain network. The blockchain has to be intended as a ledger made of blocks. Each block contains some data (e.g., transactions) and is chained to a previous and next block, so forming a chain of blocks (chain of data). The chaining mechanism takes place since each block, among the contained data, includes the hash of its previous block's content. Every transaction one makes is recorded on the public ledger once validated, with a transaction being initiated upon the triggering of various events on the blockchain, including but not limited to the transfer of cryptocurrencies (native network cryptocurrencies or ERC-20 tokens) and the invocation of smart contracts. Blockchains use different techniques to achieve the same goals of transparency, pseudo-anonymity, decentralization and tamper-proof: it is the case of Bitcoin, Zcash and Ethereum, that achieve their goals through diverse design. With the Bitcoin network, the principles and technology of blockchain have been introduced for the first time[9]. Bitcoin's purpose is to offer users the possibility of joining a cash system where cryptocurrency transfers occur through anonymous addresses without going through a financial institution. Other blockchains like Zcash and Ethereum follow the same idea, in particular *Ethereum* expands it by proposing itself as the place where dApps can run, where a dApp is a software application running in a decentralized network by exploiting the blockchain technology. DeFi is a dApp example, with the purpose of offering financial services.

2.1 ETHEREUM

Ethereum is the most used public blockchain for settling transactions [1]. It employs the account model (users store their assets in accounts), with two types of accounts available: *externally owned accounts* (EOA) and contract accounts. **EOA** is managed by an individual user via an asymmetric cryptographic key pair, consisting of a private key and a corresponding public key, exclusively held by the user. A user doesn't need to personally worry about the generation of these keys since the wallet he will use for interacting with the blockchain will automatically manage the process. A wallet can be intended as a user interface through which users can submit transactions to the blockchain. There are several types of cryptocurrency wallets, each offering different levels of security, accessibility, and convenience. Some of the most common types of blockchain wallets include software wallets (desktop, mobile or web applications) and hardware ones (physical devices specifically designed to store cryptocurrency keys offline, providing an extra layer of security by keeping the keys away from internet-connected devices). The EOA's private key enables the account owner to send signed transactions from that account, a signing to be intended as a digital signature put over the transaction, guaranteeing authentication, integrity and non-repudiation principles for it. The public key is used to derive an address for the EOA, in particular the public address corresponds to the hash of the EOA's public key and has a hexadecimal format. Such an Ethereum address can be mapped to a human-readable name through the *Ethereum Name Service* (ENS), a naming system implemented as a smart contract with the purpose of providing a more user-friendly way of transferring assets on Ethereum. **Contract accounts** are those related to smart contracts. A smart contract is a piece of code containing functions that can be triggered over the blockchain. Functions can be interpreted as the smart contract's action set. Once a smart contract is generated and published over the blockchain, it is immutable and persistent, meaning that neither its developer can tamper with it. Smart contracts are identified by their own address, generated as the hash of their contract code. Contract accounts cannot initiate transactions, but their address can be used as destination address by a transaction made by EOA: this will trigger the execution in the EVM of the contract code related to that smart contract. Transactions issued by EOA can either create a new contract account or call existing accounts (another EOA or a smart contract). The execution of a transaction comes with a cost known as *gas fees*. Such a cost is paid by the transaction issuer, who has a balance in *ether* (ETH) in the account he owns, where *ether* corresponds to the Ethereum native currency. Such a balance is altered by the transactions occurring. Generating a transaction does not mean having it inserted into the

blockchain. Before this happens, the transaction needs to be validated. In the time window between transaction generation and transaction validation, that transaction lays into a memory pool (*mempool*). Validating a transaction means inserting it into a new block of the blockchain. Blockchain does not increase in size transaction after transaction, but block after block: validating a transaction involves the addition of a new block to the blockchain, which includes the verified transaction along with others. The addition of new blocks to the blockchain adheres to the Proof-of-Work (PoW) consensus protocol, which involves the mining activity carried out by miners. Miners are individuals or entities responsible for proposing new blocks to the blockchain (*mining blocks*), determining the order of transactions within those blocks. They achieve this by validating the transactions included in the proposed block, subsequently propagating the data across the network. A miner successfully adds a new block to the blockchain by solving a mathematical problem before other miners do. It essentially becomes a competition where the winner is who comes first to the solution. Solving the mathematical problem requires computational effort. The greater a miner's computational power, the higher his chances of winning the competition. Groups of miners can combine their computational power (*mining pools*) to increase their chance of winning the competition, where winning the competition implies a reward in terms of currencies (in the event of a competition won by a mining pool, the reward would be distributed among all participants based on the proportion of computational power each of them has contributed with to the PoW protocol). The mathematical problem miners must solve in order to create a new block filled with transactions to be inserted into the blockchain involves finding that numerical value to be appended to the block's content so that the resulting hash (of the whole block content) starts with n zero-values. The n value can change to adjust the computational complexity required by the PoW protocol: the protocol has been built in such a way that, on average, a new block is created within a certain timeframe. To maintain this average, the complexity of the mathematical problem increases linearly with the available computational power. In simple terms, as more miners participate in mining, hence more computational power is allocated to the activity, greater computational effort will be required to add a new block. The miner is the one who selects, among the pending transactions present in the memory pool, those to be inserted into a new blockchain block. The selection process considers two main factors: the transaction fee associated with each transaction and the block gas limit, which specifies the maximum amount of gas units that may be consumed by dealing with all the transactions within the block (a block can contain a limited number of transactions). The transaction fee in a quantity whose amount is up to the sender (the user who issued the transaction). A higher transaction fee set by the sender increases the likelihood of the

miner including that transaction in a new block, as it results in a greater reward for the miner upon validation. This incentivizes miners to prioritize transactions with higher fees when constructing new blocks.

2.2 GAS FEES

The EVM is where code related to smart contracts deployed over the *Ethereum* blockchain is executed. Any operation pursued by the EVM (*opcode*), based on the complexity of the operation itself, has several gas units assigned. For each gas unit, there is a gas price to be paid: this is where transaction gas fees originate. Whenever the smart contract code is executed in the EVM, that execution consumes a certain amount of gas [10]. At each transaction, the sender needs to define the maximum amount of gas units the transaction is allowed to consume. This amount takes the name of gas limit. Each gas unit has a cost named gas price, whose maximum amount is settled by the user as well. The gas amount is generally expressed in Gwei (1 Gwei = 10^{-9} ETH). The maximum amount of fees a user will pay for a transaction is then given by the product of the two quantities:

$$\text{Max transaction fees} = \text{gas limit} \times \text{gas price}$$

Gas fees (or transaction fees) do not correspond to a fixed quantity, that is since they are up to the overall *Ethereum* traffic volume at the time of transaction initiation: the higher the network congestion, the higher the gas fees. Due to the blockchain's dynamic nature, one does not know statically how much gas will his transaction burn. In general nowadays, if a transaction does not consume all the gas assigned to it, then surplus gas is refunded to the caller (Section 2.2 for details); however, if a transaction runs out of gas (a transaction validation requires gas that exceeds the set gas limit), an Out-of-Gas exception is thrown by the EVM and the transaction will fail. The failed transaction would be recorded on-chain and any used gas would not be refunded to the sender. Since miners' reward is up to the transaction fees, they are naturally incentivized to insert transactions with higher gas prices into their blocks. The sender of a transaction therefore faces a trade-off between timely inclusion and cost of his transaction [11]: a higher gas price will increase the likelihood of having a transaction included quickly. To mitigate the risk of overpaying for transaction fees, gas price oracles have been developed. Among their functions, these oracles provide recommendations for the appropriate gas price required for a transaction to be included in a block within a specified timeframe. Oracles are

distinguished as on-chain and off-chain. Off-chain oracles are external entities or systems that provide data to smart contracts from outside the blockchain. These oracles can be servers, APIs, IoT sensors, or any other external data source. On-chain oracles, on the other hand, are mechanisms that operate directly within the blockchain network to provide data to smart contracts. These oracles can be implemented as smart contracts or protocols that gather and distribute data within the blockchain. Wallets utilize oracles to generate gas price suggestions for users who are initiating transactions through the wallet's interface. For a transaction to be included in a newly proposed block, its gas price has to be at least as high as the block's *BaseFeePerGas*. This value is not a constant one, it changes from block to block depending on network congestion. All the transactions within the same block are subject to the same *base fee*.

EIP1559 VS LEGACY TRANSACTIONS

Gas fee details differ according to the type of transaction sent over the blockchain. *Ethereum Improvement Proposal 1559* (EIP-1559)¹ defines a new standard around the Ethereum protocol concerning gas fee setting. Before EIP-1559, the transactions with a smaller gas fee on the Ethereum chain often remained pending for a long time because the blocks are always filled with the highest paying transactions. To eliminate this, EIP-1559 introduced a new system of gas fees with a base fee per block and a tip for the miner, corresponding to his reward for the transaction inclusion into the block. With EIP-1559, the gas limit of the blocks doubled, which means there is space for more transactions in one block. The EIP-1559 upgrade is fully compatible with previous versions, thus transactions not following the standard continue to function normally as well. This translates into two possible transaction types: Type-0 (legacy) and Type-2 (EIP-1559). The two transaction types differ in the number of parameters related to gas fees they present². Type-0 transaction has only one parameter for the gas fee setting (*gasPrice*, whose meaning is the one already discussed). Type-2 transaction has two parameters for the same purpose: *maxFeePerGas* and *maxPriorityFeePerGas*.

- *MaxFeePerGas* is the maximum amount of gas fee a user is willing to pay per unit of gas for a transaction.
- *MaxPriorityFeePerGas* is the tip a user sets for the miner. The higher is the tip, the higher will be the desire of the miner to include that transaction in the block. This parameter

¹More details concerning EIP-1559 standard here: <https://www.quicknode.com/guides/ethereum-development/transactions/how-to-send-an-eip-1559-transaction>

²For details over gas fees related parameters:<https://www.quicknode.com/guides/ethereum-development/transactions/how-to-send-transactions-on-ethereum-using-python>

determines transaction priority.

Once the parameters related to gas fees are set for a transaction, the amount of fees a user will spend at most is so retrievable:

$$\text{Max transaction fees} = \text{gas limit} \times \text{gas price}$$

where:

- In Type-0 transaction, *gas price* is the price per gas unit set by the sender, including the miner's reward;
- In Type-2 transaction, $\text{gas price} = \min(\text{BaseFeePerGas} + \text{maxPriorityFeePerGas}, \text{maxFeePerGas})$.

The transaction fees so defined correspond to the amount of fees burnable at most. Considering the *gasUsed* as the effective amount of gas unit burnt by a transaction:

- In case of Type-0 transactions, once confirmed, the amount truly burnt overall corresponds to $\text{transaction fees} = \text{gasUsed} \times \text{gas price}$,

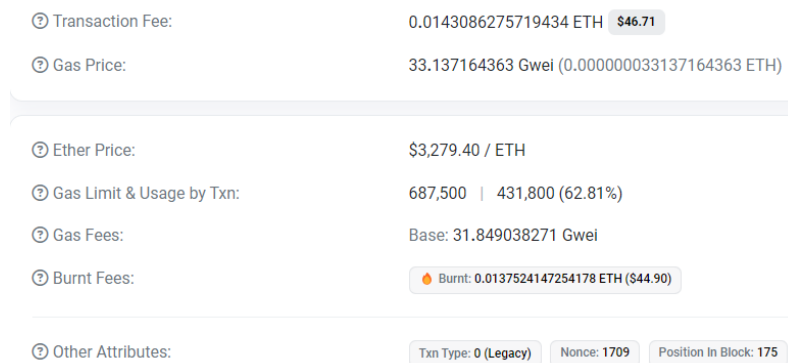


Figure 2.1: Legacy transaction on Etherscan.

- In case of Type-2 transactions, once it is confirmed, the quantity truly burnt is $\text{transaction fees} = \text{gasUsed} \times \text{gas price}$ with *gas price* defined as above and the amount $(\text{MaxFeePerGas} - (\text{BaseFeePerGas} + \text{maxPriorityFeePerGas})) * \text{gasUsed}$ is refunded back to the sender of the transaction itself (*saved fees*).

Gas fees are what is burned anytime a smart contract function is triggered. This is the case of Tornado Cash as well.

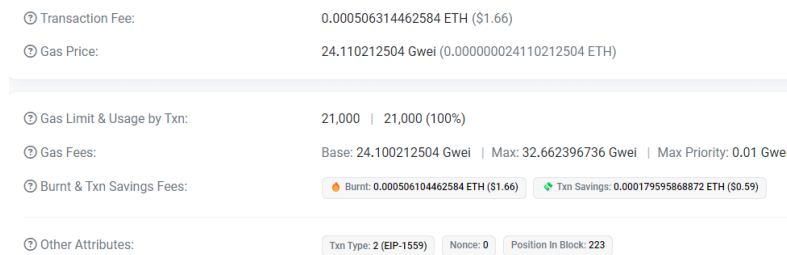


Figure 2.2: EIP-1559 transaction on Etherscan.

2.3 TORNADO CASH

The Ethereum account model has several implications from a privacy standpoint. Firstly, it incentivizes the reuse of accounts across multiple transactions, so facilitating the profiling of transaction histories. To address this issue, users can employ a mixer, a tool designed to protect the privacy of blockchain addresses by breaking the link between an address and its transaction history. TC is a dApp belonging to the family of coin mixers and operating across multiple networks using smart contracts. These networks include Ethereum Mainnet, Binance Smart Chain, Polygon, Optimism, Arbitrum, Gnosis, and Avalanche Mainnet. Due to the high level of TC usage activity on the Ethereum Mainnet, the network is the main focus of this project. Within the family of coin mixers, TC is categorized as a non-custodial one. Mixers can be custodial or non-custodial [1]:

- In a custodial mixer, users send their coins to a trusted party, who in return sends back “clean” coins after some timeout. During mixing a user does not retain ownership of his coins, hence the trusted mixing party might steal funds.
- A non-custodial mixer replaces the trusted mixing party of custodial ones with a publicly verifiable smart contract. Non-custodial mixing is a two-step procedure. First, users deposit equal amounts of ether or other tokens into a mixer contract from address \mathcal{A} . After some user-defined time interval, they can withdraw their deposited coins with a withdrawal transaction to a fresh address \mathcal{B} . In the withdrawal transaction, users can prove to the mixer contract that they deposited without revealing which deposit transaction was issued by them by using one of several available cryptographic techniques (zkSNARK in TC context).

TC performs its job through the usage of smart contracts. Each contract has an address and can be thought as a pool where currencies can be deposited in and withdrawn from. What TC, and other mixers, do is “mixing” a user transaction with those of others in a pool, making it

harder to link deposits and withdrawals from that pool. Each single pool accepts a fixed amount of the same currency.

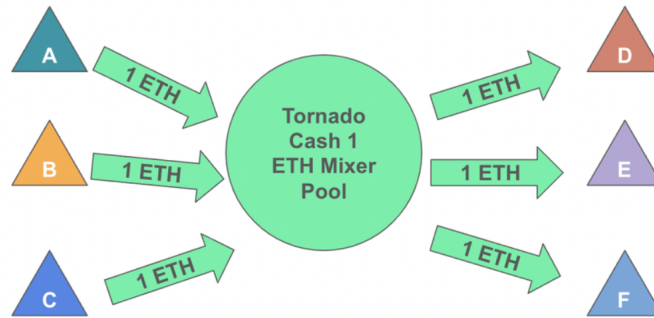


Figure 2.3: Example of the Tornado Cash 1 ETH pool: addresses A through F deposit to and withdraw from the pool. It quickly becomes impossible to associate withdraw and deposit transactions given a growing mixer [1].

Currencies accepted by the TC platform in the case of its usage over the Ethereum Mainnet network correspond to the native one (ETH) plus several Ethereum-based tokens (e.g., *DAI*, *USDC*, *wBTC*, etc). Each currency has four pools to it related. Among the currencies, ETH is the most frequently used for transactions within the Tornado Cash ecosystem on the Ethereum network, thus warranting attention in this project. In TC, users are required to complete the coin mixing in two steps: deposit and withdraw. Users deposit equal amounts to a TC smart contract (pool). After some time, users can withdraw their funds from the mixer contract to a freshly generated EOA by providing a zero-knowledge proof (ZKP) that proves that the withdrawing user is one of the depositors. At this point, the withdrawing EOA has enhanced its privacy since it has become unlinkable to any unique depositor EOA. A user’s anonymity is defined by the number of equal user deposits in a given pool. This is the pool’s *Anonymity Set*: the more users deposit in the pool, the greater the number of people that a withdrawal could come from [1]. The definition of anonymity set implies that any of its members are equally likely to be the deposit address actually linked to a given withdrawal address.

2.3.1 DEPOSIT

Using Tornado Cash means performing a deposit with respect to one of its pools (smart contracts). Each currency has its own pools. Focusing on the native currency of the Ethereum

mainnet network (ETH), there are four pools a user can deposit into. Each of these pools can be subject to the deposit of a fixed amount of ether: 0.1 ETH, 1 ETH, 10 ETH, 100 ETH.



Figure 2.4: TC pools for ETH currency in the Ethereum mainnet network.

If a user wants to deposit 111 ETH, he has to perform a total of three different deposits:

- One deposit towards the smart contract related to the fixed amount of 1 ETH;
- One deposit towards the smart contract related to the fixed amount of 10 ETH;
- One deposit towards the smart contract related to the fixed amount of 100 ETH.

To do so, the depositor has to interact with the related smart contracts, hence with the related blockchain addresses.

ETH amount	Smart contract address	Created time (+UTC)
0.1	12D6f87A04A9E220743712cE6d9bB1B5616B8Fc	2019-12-16 19:08:43
1	47CE0C6eD5B0Ce3d3A51fdb1C52DC66a7c3c2936	2019-12-16 22:17:53
10	910Cb523D972eb0a6f4cAe4618aD62622b39DbF	2019-12-16 22:46:55
100	A160cdAB225685dA1d56aa342Ad8841c3b53f291	2019-12-25 18:02:56

Figure 2.5: TC smart contracts addresses for ETH currency in the Ethereum mainnet network [2].

At deposit time, the depositor generates two random numbers $k, r \in \mathbb{B}^{248}$ with $B = \{0, 1\}$, k a n -bit nullifier and r a n -bit randomness. Both values are known only to the depositor and

must remain secret at all times. The depositor hashes the concatenation of the generated values through the Pedersen hashing function H_1 [12], so obtaining the commitment $C = H_1(k||r)$ as a 256-bit unsigned integer. The pre-image of the commitment (hence, the concatenation of the two generated random values) corresponds to a secret *note* saved on the depositor side in the form of a *.txt* file. The *note*, corresponding to a sequence of digits, will be needed for the withdrawal purpose.



Figure 2.6: Example of TC secret note, received after a deposit in the Goerli testnet network.

The commitment so made ($H_1(\textit{note})$) is inserted into a data structure called Merkle Tree as a new non-zero leaf [13].

MERKLE TREE

A Merkle Tree is a complete binary tree structure in which each leaf node is a hash of a block of data (Pedersen hash of the *note* in TC), and each non-leaf node is a hash of its children (MiMC as used hash function in TC).

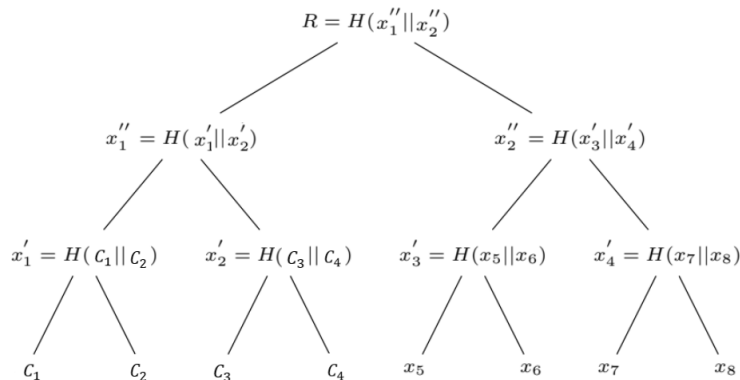


Figure 2.7: Toy example of a TC Merkle tree with height 3, four deposits C_i and four pre-initialized zeros leaf x_i .

In TC context, the Merkle tree has height 20, with 2^{20} possible leaves for a single Merkle tree, hence 2^{20} possible deposits. Each TC pool (smart contract) has its own Merkle Tree. When the Merkle tree related to a single pool is full, a new one is needed. The Merkle tree is initialized

with all leaves being a default value, called zero-leaf. Whenever a new deposit is performed towards a specific pool, the related Merkle tree leaf content, starting from the most left zero-leaf, is replaced with the commitment value. By then, the content of all the nodes in the path from that node to the root is updated as well [14]. The sequence of neighboring nodes required to update the Merkle tree root after the addition of a new leaf is referred to as the Merkle path³.

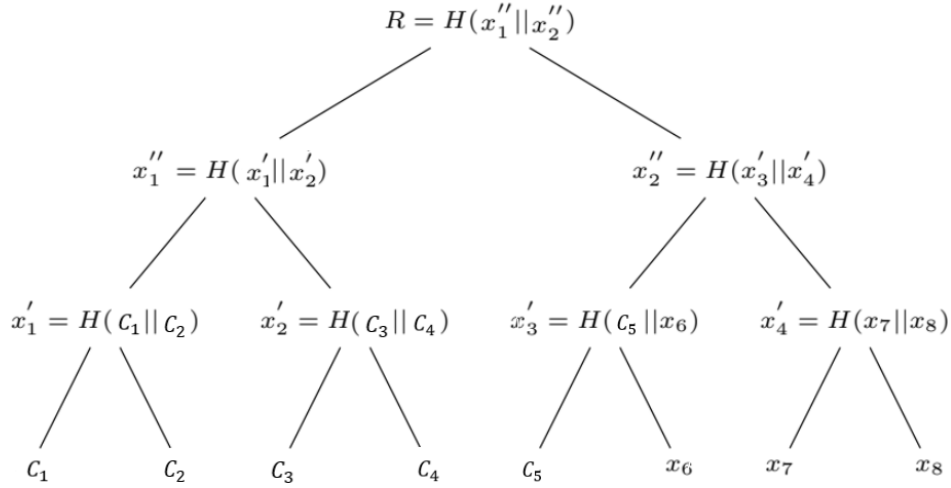


Figure 2.8: Toy example of a TC deposit in the Merkle tree (c_5), with related Merkle path $\{x_6, x'_4, x''_1\}$

2.3.2 WITHDRAW

When a user considers the size of the anonymity set as satisfactory, he may proceed to withdraw his asset from the mixer. To do so, the withdrawer submits the *note* generated at deposit time and the recipient address to the TC platform. Once the related smart contract completes the zkSNARK proof based on the *note*, through which the withdrawer proves to know the pre-image of a previously inserted hash leaf in the Merkle tree without revealing the leaf itself, the amount left after deducting the fee required for the smart contract operations is transferred to the corresponding receiver.

zkSNARK

zk-SNARK is a protocol that lets one party, the prover, prove to another party, the verifier, that a statement about some privately held information is true without revealing the informa-

³For details: https://iden3-docs.readthedocs.io/en/latest/iden3_repos/research/publications/zkproof-standards-workshop-2/merkle-tree/merkle-tree.html

tion itself [15]. zkSNARK stands for zero-knowledge, succinct, non-interactive argument of knowledge. A parsing of the single terms follows [16]:

- **Zero-Knowledge:** The proof is said to be zero-knowledge if it does not reveal the secret value or any other information besides the proof that a public statement is true. In the context of Tornado Cash, a user will be able to construct a proof that he has previously deposited to the Tornado Cash contract without having to reveal the specific deposit transaction;
- **Non-Interactive:** The proof does not require any direct interaction between the prover and the verifier. In other words, a single message from the prover to the verifier is sufficient;
- **Succinct:** The proof can be efficiently verified with respect to data size and verification runtime (large storage data or complex on-chain computations would be infeasible in the blockchain context);
- **Argument of Knowledge:** The knowledge of the secret value the proof is built around.

The protocol in TC takes as input a public statement and a secret one so made:

- $PublicStatement = (root, nullifier, recipientAddress)$, with:
 - Root: One of the recent roots of the Merkle Tree, considering that a TC smart contract saves in its state the history of its last 100 roots [13];
 - Nullifier: the Pedersen hash of the k -secret value;
 - Recipient address: the blockchain address where funds should be sent to.
- $SecretStatement = (k, r, commitment, MerkleProof(commitment))$, with:
 - k : the secret nullifier random value generated at deposit time, it is part of the saved *note*;
 - r : the secret randomness value generated at deposit time, it is part of the saved *note*;
 - Commitment: the Pedersen hash of the saved *note*;
 - MerkleProof(commitment): the Merkle path of the leaf corresponding to the specified commitment.

A Circom circuit will verify that the secret statement is consistent with the public one, if so the withdrawal is considered as legitim and the related amount can effectively be sent to the address signaled as the recipient one in the public statement. In particular, the circuit will verify that:

- The MerkleProof is valid, hence the leaf of interest is truly part of the Merkle Tree whose recent root is the one specified in the public statement. This validation is achieved by calculating the root that the Merkle Tree would have if the provided Merkle path (Merkle proof) was genuinely incorporated into the Merkle Tree. If the computed root matches the recent root provided in the public statement, the validation yields a positive result;
- The Pedersen hash of the *note* (k, r) truly corresponds to the given commitment;
- The nullifier in the public statement truly corresponds to the Pedersen hash of the private value k , part of the *note*.

Checks made by the circuit aim to make sure that

- Private statement is consistent with the public one;
- Provided nullifier in the public statement has not been submitted before. This prevents from the attempt to perform a withdrawal providing the same note more than once (*double-spent problem*).

RELAYER

A withdrawal from a TC smart contract can be triggered in two different ways [17]:

- The user uses a relayer to make the withdrawal to any Ethereum recipient address without needing to make the wallet connection on the Tornado Cash website. Since the relayer is in charge of paying for the transaction gas, he will receive a small portion of the deposit for both a refund and a reward for his job.
- The user links their wallet (Metamask or WalletConnect) to the Tornado Cash website, and they pay for the gas needed to withdraw the amount deposited.

A relayer is an independent operator that provides an optional service for Tornado Cash users to help them solve the *fee payment dilemma*. This dilemma pertains to the challenge of covering the costs (fees) associated with withdrawing from a mixing pool while preserving

anonymity: being able to cover the withdrawing costs translates into having funds associated with the address requiring the withdrawal, hence having a transaction history. A relay can trigger a withdrawal transaction instead of the user, sending the withdrawal amount to a new account with no ETH balance (*fresh address*) and deducting the withdrawal fee (both for the paid gas fee and a reward for the performed action) directly from the transfer amount. In case a user is interested in the relay figure along a withdrawal, he needs to select a relay to create a withdrawal transaction. The relay transaction creates two internal transactions: one withdraws the transaction fee from the mixing pool to the account of the relay, while the other transfers the remaining funds to the account of the user [3].

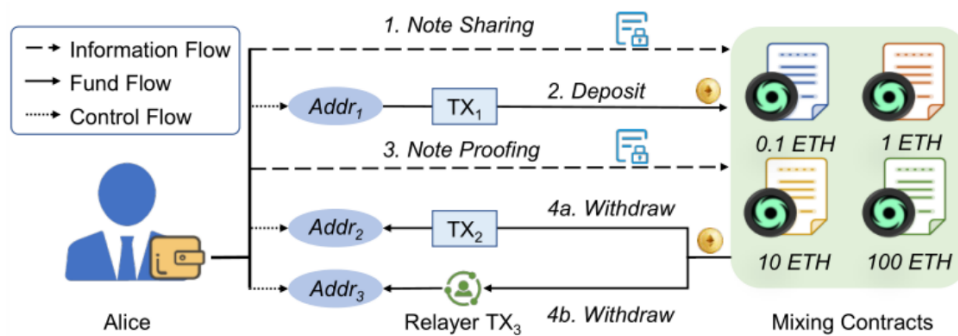


Figure 2.9: TC general working schema [3].

To give a concrete example of relay employment in TC (TX_2 and TX_3 in Figure 2.9), a real TC withdrawal taken from Etherscan blockchain explorer is here reported:

From:	reitor.eth
To:	0xd90e2f925DA726b50C4Ed8D0Fb90Ad053324F31b (Tornado.Cash: Router)
Value:	0 ETH (\$0.00)
Transaction Fee:	0.0238933779442636 ETH (\$79.36)
Gas Price:	55.334363002 Gwei (0.000000055334363002 ETH)

Figure 2.10: Example of a withdrawal from 0.1 ETH pool involving a relay.

In Figure 2.10, the meanings of the assigned attributes are the following:

- **From** is the relay who performs the withdrawal on behalf of the user whose address is the one specified in the first transaction in **To**. In this case, the relay makes use of the ENS.

- **To** is where the two transactions triggered by the relayer are highlighted: the first transaction is the one where the amount transferred from the smart contract to the recipient address is specified (TX_2 in Figure 2.9), the second transaction is the one specifying the amount transferred from the smart contract to the relayer address (TX_3 in Figure 2.9). The amounts transferred along the two transactions sum up to 1 ETH (the amount deposited). The amount received by the relayer covers both the withdrawal's **Transaction Fees** and the reward for the performed proxy action, corresponding to a percentage of the withdrawn amount. The amount transferred to the relayer is directly taken from the amount sent to the recipient.

2.4 USEFUL DEFINITIONS

To point the attention to specific concepts the thesis project core deals with, a recapitulation of some definitions outlined in the background overview are here provided, supplemented by additional ones considered pertinent for the reader's comprehension:

- **Etherscan**: an Ethereum blockchain browser that offers access to transaction informations.
- **Web3**: a library that provides developers with a convenient way to interact with the Ethereum network.
- **Infura**: a service that allows developers to interact with the Ethereum blockchain without needing to run their own local Ethereum node. Developers can use *infura* as an access point to interact with the *Ethereum* blockchain. In order to do so, a private key given by *infura* itself is needed.
- **Anonymity set**: the level of privacy of a specific pool (smart contract) belonging to the TC platform. It translates into the number of users who has deposited currencies into that pool.
- **Ethereum Name Service**: a decentralized service based on Ethereum, which translates blockchain domain names (e.g., *Relayer.eth*) to blockchain addresses.
- **Anonymity mining**: From December 2020 to December 2021 [18], TC started offering a reward in anonymity points (AP) that could be exchanged in TORN (TC native currency) to users employing TC. The reward amount used to be up to the deposited amount and duration (the time period that amount was left into a TC pool). The aim of such a reward was to induce more users into making deposits in TC pools, thereby increasing their anonymity sets.

- **Gas price:** the amount of Gwei a user will pay per gas unit for the initiated transaction.
- **Gas limit:** the maximum amount of gas unit a user is willing to pay for his transaction.
- **MaxFeePerGas:** In EIP-1559 transactions, is the maximum amount of Gwei a user is willing to pay per unit of gas. It includes the *MaxPriorityFeePerGas* value.
- **MaxPriorityFeePerGas:** In EIP-1559 transactions, it is the amount of gas reserved as a reward for the miner.
- **BaseFeePerGas:** A block-related parameter that establishes the minimum gas price paid by all the transactions validated in that block.
- **WalletConnect:** a protocol that enables interaction between *dApps* and mobile wallets, without compromising their private keys.

3

Related works

In Section 2.3 insights of the Tornado Cash platform have been given, in particular the anonymity set, corresponding to the level of privacy guaranteed to a user who is willing to perform a deposit towards a specific pool, has been defined as the number of equal user deposits in that pool. Careless TC usage tends to reveal links between deposits and withdraws, impacting the anonymity of other users. That is since if a deposit can be linked to a withdrawal, it will no longer truly contribute to the claimed anonymity set. Ethereum privacy in terms of address correlation is a research concern. In general, existing address correlation methods on Ethereum involve two major categories [2]. One is using machine learning and node embedding methods to cluster transaction behavior patterns or user accounts with similar characteristics [19][20][21], the other is using heuristic or graph-based clustering algorithms to link addresses that participated in certain transactions[22]. In the TC optic, starting from on-chain data, [2] proposes three heuristic clustering rules to achieve address correlation for Tornado coin mixing transactions based on the time interval features:

- **Heuristic 1:** Given a deposit d , a withdrawal w and a time interval between the two δ , if $\delta \leq 180s$ and both d and w refer to the same TC smart contract (pool), then the addresses the deposit started from and the withdrawal amount will be delivered to belong to the same user;
- **Heuristic 2:** Given multiple single deposits and withdrawals related to the same TC pool $\{\langle d_1, w_1, \delta_{dw_1} \rangle, \langle d_2, w_2, \delta_{dw_2} \rangle, \dots, \langle d_n, w_n, \delta_{dw_n} \rangle\}$ with δ_{dwi} corresponding to the time interval between d_i and w_i , when $n \geq 2$, if $\forall \{d_i, d_{i+1}\} \subseteq \{d_1, d_2, \dots, d_n\}, \{w_i, w_{i+1}\}$

$\subseteq \{w_1, w_2, \dots, w_n\}$, $\delta_{wd} = d_{i+1}.timestamp - w_i.timestamp$ one of the following conditions is satisfied:

- $d_i.from = d_{i+1}.from$, and $\delta_{dwi}, \delta_{dw(i+1)} \leq 20 \text{ min}$, $\delta_{wd} > 0$, with $d_i.from$ corresponding to the address who triggered that deposit;
- $w_i.input.recipient = w_{i+1}.input.recipient$, and $\delta_{dw(i+1)} > 0$, $\delta_{dwi} \leq 20 \text{ min}$, $\delta_{wd} \leq 20 \text{ min}$, with $w_i.input.recipient$ corresponding to the address that is asking for the amount to be withdrawn;
- $d_i.from = d_{i+1}.from$, and $w_i.input.recipient = w_{i+1}.input.recipient$, $\delta_{wd}, \delta_{dwi}, \delta_{dw(i+1)} > 0$

then the addresses all the deposits involved started from and all the withdrawals involved are directed to in these transactions belong to the same user.

- **Heuristic 3:** Given a set $\mathcal{D} = \{d_1, d_2, \dots, d_n\}$ of $n > 2$ deposits and a set $\mathcal{W} = \{w_1, w_2, \dots, w_n\}$ of $n > 2$ withdrawals with a time interval between the occurring of the last deposit in \mathcal{D} ($d_n.timestamp$) and the first withdrawal in \mathcal{W} ($w_1.timestamp$) equal to Δ ($\Delta = w_1.timestamp - d_n.timestamp$), if it simultaneously happens that:

- All the deposits of the first set (\mathcal{D}) have been triggered by the same address,
- All the withdrawals of the second set (\mathcal{W}) have the same address recipient,
- $\delta_d, \delta_w \leq 10 \text{ min}$ and $\Delta \leq n \times 12 \text{ h}$, with $\delta_d = \max\{d_{i+1}.timestamp - d_i.timestamp \mid d_i, d_{i+1} \in \mathcal{D}\}$, $\delta_w = \max\{w_{i+1}.timestamp - w_i.timestamp \mid w_i, w_{i+1} \in \mathcal{W}\}$

then the addresses all the deposits involved started from and all the withdrawals involved are directed to in the n transactions belong to the same user.

Authors of [1] increase the number of heuristics related to the same purpose. They propose a tool named Tutela, funded by the Tornado Cash community itself. The application combines five heuristics (state-of-the-art heuristics plus new proposed) to compute a true anonymity set for each TC pool. The heuristics in place correspond to:

- **Heuristic 1:** Suppose the address making a deposit transaction to a Tornado Cash pool matches the address making a withdrawal transaction from the same pool. In that case, the two transactions can be linked.

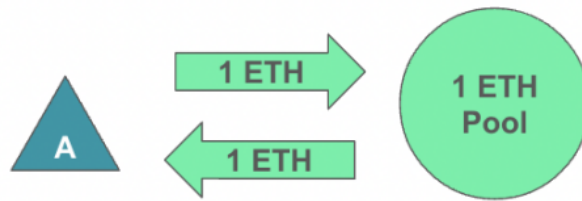


Figure 3.1: Heuristic 1 schema: a single address A withdrawing and depositing to the same TC pool [1].

- **Heuristic 2:** Suppose the address making a deposit transaction to a Tornado Cash pool specifies a custom-set gas price that perfectly matches the one specified by the address making a withdrawal transaction from the same pool. In that case, the two transactions can be linked.

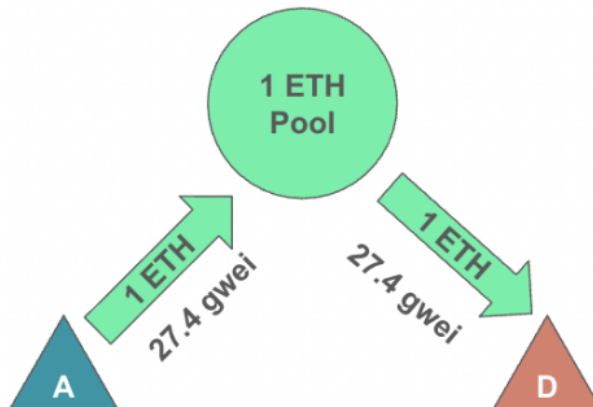


Figure 3.2: Heuristic 2 schema: two addresses (A and D) depositing to and withdrawing from the same TC pool with an equal custom-set gas price [1].

- **Heuristic 3:** This heuristic aims to link withdrawal and deposit transactions on Tornado Cash by inspecting ETH non-Tornado Cash interactions. This is done by constructing two sets, one corresponding to the unique Tornado Cash deposit addresses and one to the unique Tornado Cash withdraw addresses, to then make a query to reveal transactions between addresses of each set: when at least three such transactions are found for a pair, the withdrawal and deposit addresses will be considered heuristically linked in Tornado Cash. The more transactions are found, the more confident the link.

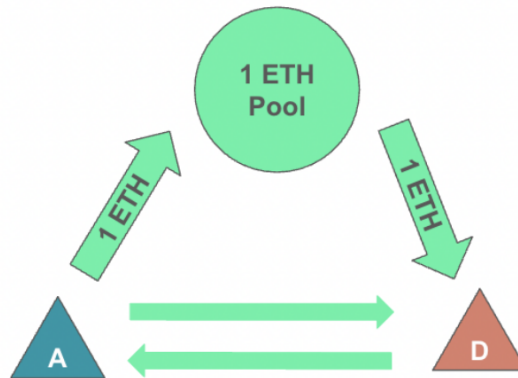


Figure 3.3: Heuristic 3 schema: addresses A and D deposit and withdraw from the same TC pool, moreover interactions in terms of transactions do exist between them out of TC [1].

- **Heuristic 4:** The portfolio of an address' withdrawals across Tornado Cash pools is studied. Then the point is to search for all addresses whose portfolio of deposit transactions is exactly the same as the first address' withdrawal portfolio. To put it simply: the heuristic looks for two addresses A and D who deposit and withdraw the same number of times from the same Tornado Cash pools.

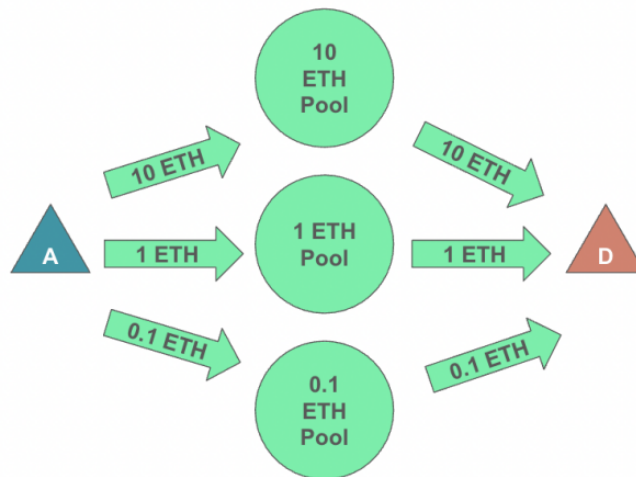


Figure 3.4: Heuristic 4 schema: addresses A and D deposit and withdraw the same number of times from the same three Tornado Cash pools [1].

- **Heuristic 5:** Thanks to anonymity mining, after withdrawing assets, users could claim anonymity points. Due to the reward dependency on the deposit amount and period,

one can calculate the Ethereum blocks that separate the deposit and withdrawal transactions of that asset. If there is a unique deposit/withdrawal combination in a pool separated by the calculated number of Ethereum blocks, the transactions are assumed linked. Because of the ending of the anonymity mining program in December 2021, this heuristic does not hold for TC transactions made after that.

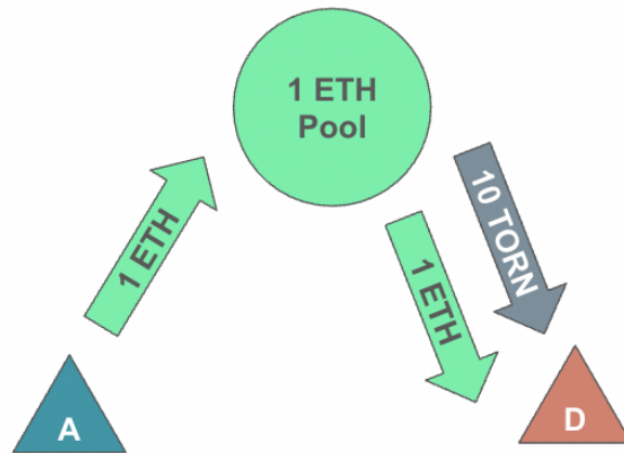


Figure 3.5: Heuristic 5 schema: address D performs a withdrawal from the 1 ETH pool, obtaining a reward implying that the deposit has been in the pool for n blocks. The only deposit present in the prior n block is the one made by A , so the two addresses are linked [1].

According to the five mentioned heuristics, authors of [1] identified 42.8k potentially compromised transactions over 97.3k Tornado Cash user deposits. Splitting the result by pools translates into a reduction of the *anonymity set* of the time by 37%.

Some of the explored heuristics have been used in [10] to build a ground truth in order to measure how well their proposed technique based on time-of-day transaction activity, gas price distribution and transaction graph analysis identifies the linked withdraw-deposit address pairs. Evaluation on heuristically linked mixing participants showed that profiling techniques, especially node embedding algorithms, can reduce the anonymity set sizes of the mixing parties.

Authors of [3] tackle the mixing address correlation problem using graph feature learning technique. They have first built an interaction graph whose vertices represent user accounts and links represent mixing transactions. They have then designed a GNN-based link prediction mechanism, which can automatically extract deeper neighborhood features from the interaction graph, and create new links between accounts by mapping them to different representations in embedding space, hence calculating the probability of correlation between account

nodes through node embeddings. With their approach, the mixing transaction address correlation problem has been transformed into a link prediction task. According to their experiment, their technique allows to improve the correlation score over the state-of-the-art methods.

In [7] it is pointed out that anonymity mining does not necessarily improve the quality of a mixer's *anonymity set*: the reward attracts privacy-ignorant users with a primary interest in mining rewards, who then do not contribute to truly improving the privacy of other mixer users since they can be told apart through heuristics. In particular, authors have empirically shown that after the introduction of anonymity mining, the number of users who reuse the same address for both deposits and withdrawals has increased, leading to a rise in the capability of the *address reuse* heuristic (Figure 3.1) in terms of reduction of the anonymity set from 7% (before AM launch) to 13.5% (after AM launch) on average.

The related works have explored some platform usage patterns that can be used to link deposits and withdraw, revealing that TC's proposed anonymity sets are mostly inaccurate. Apart from showing the inaccuracy of the TC anonymity set, the need to break the anonymity of TC by correlating back transaction addresses comes from Tornado Cash's involvement in cybercrime activities, such as money laundering.

Related works converge in a single concern: immature user behavior in Tornado Cash prevents it from achieving its highest attainable privacy guarantee. The explored heuristics appear to be reliant on the user's behavior within the TC mixing scenario. Due to their simplicity, there may be occurrences of false positives in practice (addresses wrongly grouped in a cluster or untrustworthy links) [1]. Approaches that reveal connections between deposits and withdrawals in presence of a privacy-conscious usage of the TC platform by its users have not been thoroughly investigated yet.

4

Project core

On any public blockchain, the cost of creating a new EOA is virtually zero, enabling the same entity to manage several pseudonymous addresses from which transactions can start[1]. A new transaction inserted into the blockchain implies the burning of the related transaction fees, where fees could be suggested by the wallet itself. Wallets provide customers with the ability to send and receive virtual currency, tuning their balance through interaction with blockchains. Unlike traditional pocket wallets, cryptocurrencies are not stored in the crypto wallets. Cryptocurrencies are neither stored in any single area nor exist anywhere in any bodily form, but exist as data of transactions stored on the blockchain. Wallets facilitate user to create an account, i.e. a pair of private key and public key stored in a wallet software. Wallets are categorizable in software or hardware wallets. Software wallets are downloadable desktop or mobile software programs, as well as web applications. Hardware wallets are physical devices like USB drives. Wallets belonging to different categories differ in the way the EOA-related key pair is managed[23]. Within the Ethereum ecosystem, users have the option to choose from a variety of wallets. At transaction time, gas fees are typically suggested to the user by wallets according to specific algorithms. The objective of the thesis is to delve into the source code of various wallets in order to examine the algorithms they employ for suggesting gas fees. The ultimate goal is to establish a connection between transactions on the blockchain and the specific wallet from which they originated. This approach, that can be encapsulated by the term *wallet fingerprints*, hides:

- A privacy concern over the Ethereum blockchain;
- The potential to reduce the *anonymity set* of Tornado Cash pools [1]: a withdrawal transaction initiated with wallet software X will only be indistinguishable from deposit transactions initiated by users employing the same wallet software X .

The thesis moves under the assumption that if a user A performs a deposit through the wallet software X (e.g., Metamask), the same wallet software will be utilized during the withdrawal process.

4.1 WALLET ANALYSIS

Among the Ethereum-compatible wallets, according to the purpose of the project, attention has been put on those respecting some specific requirements. In particular, wallets taken into consideration are:

- Software wallets: considered more user-friendly, consequently attracting a larger number of users;
- Open source: allowing their source code analysis, necessary for the identification of the algorithm in use for the gas fee suggestions;
- *WalletConnect* compatible: wallet supporting the *WalletConnect* protocol are those usable within the TC dApp.

Among those wallets respecting the above-mentioned requisites, those analyzed in the current project correspond to: *Metamask*, *Trust Wallet*, *OneKey*, *Rainbow*, *Unstoppable Wallet* and *ShapeShift Wallet*. For each wallet, the source code related to the same pattern has been checked: retrieval of gas fee suggestions at the time of ETH currency transfer.

4.1.1 METAMASK

MetaMask is a software wallet available as both a web extension (compatible with browsers such as *Chrome*, *Firefox*, *Edge*, and *Opera*) and a mobile application (for *Android* and *iOS*). With over 10 million downloads on the *Android* platform alone, its widespread adoption serves as a testament to its popularity. It allows several operations, among which:



Figure 4.1: Metamask features at frontend level.

- *Buy*: allows a user to purchase new currencies by selecting a provider. Payment methods are up to the provider and could include debit and credit cards, *Apple pay*, *Google pay* and more. Each provider proposes personalized transaction fees for the buying operation.
- *Sell*: allows a user to sell his cryptocurrencies.
- *Send*: allows a user to perform the transfer of a certain amount of his asset towards a specified recipient address. It consists of the transfer operation.
- *Swap*: allows a user to swap his cryptocurrencies into another token (e.g., from ETH to MATIC).
- *Bridge*: allows a user to move his funds from a blockchain network to another one (e.g., from *Ethereum* to *Polygon*), with the possibility of asking for a swap along the transfer.

The sending operation is the one involving cryptocurrency transfer from one EOA to another one. At the frontend level, along with an ETH transfer, *Metamask* prompts the user with three different gas fee suggestion levels: *low*, *market* and *aggressive*. The higher the level (from *low* to *aggressive*) the higher the gas fee suggestion (in terms of *MaxFeePerGas* and *MaxPriorityFeePerGas* for type-2 transactions or *gasPrice* for type-0 transactions), hence the higher the priority attributed to the transaction. The wallet allows its users to customize the gas fees. The source code¹ of the *Metamask* wallet present on GitHub has been analyzed, looking for that snippet concerned with the assignment of values to the gas fee parameters for the three presented levels during a *send* operation. The entry point of the function related to the ETH transfer has been identified at first, localized in the packages path *metamask-extension/ui/components/app/wallet-overview/eth-overview.js*² as a *javascript* file.

¹Metamask open source code: <https://github.com/MetaMask>

²For details: <https://github.com/MetaMask/metamask-extension/blob/develop/ui/components/app/wallet-overview/eth-overview.js>

```

onClick={() => {
  trackEvent({
    event: MetaMetricsEventName.NavSendButtonClicked,
    category: MetaMetricsEventCategory.Navigation,
    properties: {
      token_symbol: 'ETH',
      location: 'Home',
      text: 'Send',
      chain_id: chainId,
    },
  });
  dispatch(
    startNewDraftTransaction({ type: AssetType.native }),
  ).then(() => {
    history.push(SEND_ROUTE);
  });
}}

```

Figure 4.2: Metamask entry point for the send transaction button.

From Figure 4.2 (frontend side), the flow of the internal called functions has been followed, crossing the connection between frontend and backend. The function identified as the one being in charge of computing and updating the gas fee values for the three priority levels is the one defined in the *typescript* file *core/packages/gas-feecontroller/src/determineGasFeeCalculations.ts*³. What the function does is checking the type of transaction (type-0 or type-2) and delegating the gas fees computation accordingly. In particular:

```

if (isEIP1559Compatible) {
  let estimates: GasFeeEstimates;
  try {
    estimates = await fetchGasEstimates(fetchGasEstimatesUrl, clientId);
  } catch {
    estimates = await fetchGasEstimatesViaEthFeeHistory(ethQuery);
  }
}

```

Figure 4.3: Metamask's code flow in case of gas fee suggestions for type-2 transactions.

```

} else if (isLegacyGasAPICompatible) {
  const estimates = await fetchLegacyGasPriceEstimates(
    fetchLegacyGasPriceEstimatesUrl,
    infuraAPIKey,
    clientId,
  );
}

```

Figure 4.4: Metamask's code flow in case of gas fee suggestions for type-0 transactions.

In case of a type-2 transaction (hence EIP-1559 compatible, Figure 4.3) a *try-catch* block follows. In the *try* branch, an HTTPS request is performed towards the URL specified as parameter (*fetchGasEstimatesUrl*). This translates into an API call. The specific URL value is up to the live setting of an environment variable, opening the way to two possible assignments for

³For details: <https://github.com/MetaMask/core/blob/main/packages/gas-fee-controller/src/determineGasFeeCalculations.ts>

it. Being the environment variable value unknown a priori, both the possible values are taken in consideration for the purposes of the project. This translates into two possible HTTPS requests (API calls) to pay attention to in the *try* branch, among which only one will be truly made:

- <https://gas.api.cx.metamask.io/networks/1/suggestedGasFees>
- <https://gas.uat-api.cx.metamask.io/networks/1/suggestedGasFees>

The numerical value present in both the URLs refers to the *chain-id* of the network of interest, with the *chain-id* corresponding to the unique identifier of a blockchain network. In this case, the integer value appearing is the 1 integer, referring to the *Ethereum Mainnet* network. In Figure 4.3's *catch* branch, several *Remote Procedure Calls* (RPCs) are performed. The most relevant is the *eth_feeHistory*⁴ RPC. The method returns a collection of historical gas information related to a sequence of blocks of interest, taking as input: an integer value representing how many sequential blocks are we interested in, the highest block number in the sequence, an optional array of percentiles. *Eth_feeHistory* method, for each block it is considering, will first sort all transactions by the priority fee. It will then go through each transaction and add the total amount of gas paid for that transaction to a bucket which maxes out at the total gas used for the whole block. As the bucket fills, it will cross percentages which correspond to the percentiles. Whenever a specified percentile in the optional input array is reached, the priority fees of the first transactions that cause it to reach those percentages will be recorded. The recorded priority fees represent the priority fees of transactions at key gas-used contribution levels, where earlier levels have smaller contributions and later levels have higher contributions [24]. Results of the method include the *baseFeePerGas* of each block of interest as well. In *Metamask* scenario, the history of the last five newest blocks is considered, with priority fees taken from each block corresponding to those at percentiles [10, 20, 30]. Results of the method invocation are formatted and, lastly, given as input to the *calculateEstimatesForPriorityLevel*⁵ function, defined in the *typescript* file present in the path of the packages *core/packages/gas-fee-controller/src/fetchGasEstimatesViaEthFeeHistory/calculateGasFeeEstimatesForPriorityLevels.ts*. The function, based on the *eth_feeHistory* formatted output (*blocks* in Figure 4.5), performs some computations to generate gas fee suggestions for each priority level (*low*, *market*, *aggressive*).

⁴For details: <https://docs.alchemy.com/reference/eth-fee-history>.

⁵For details: <https://github.com/MetaMask/core/blob/8769bd80eb9a131f9fb75ae5f85491eedfc19e62/packages/gas-fee-controller/src/fetchGasEstimatesViaEthFeeHistory/calculateGasFeeEstimatesForPriorityLevels.ts>

```

function calculateEstimatesForPriorityLevel(
  priorityLevel: PriorityLevel,
  blocks: FeeHistoryBlock<Percentile>[],
): Eip1559GasFee {
  const settings = SETTINGS_BY_PRIORITY_LEVEL[priorityLevel];

  const latestBaseFeePerGas = blocks[blocks.length - 1].baseFeePerGas;

  const adjustedBaseFee = latestBaseFeePerGas
    .mul(settings.baseFeePercentageMultiplier)
    .divn(100);
  const priorityFees = blocks
    .map((block) => {
      return 'priorityFeesByPercentile' in block
        ? block.priorityFeesByPercentile[settings.percentile]
        : null;
    })
    .filter(BN.isBN);
  const medianPriorityFee = medianOf(priorityFees);
  const adjustedPriorityFee = medianPriorityFee
    .mul(settings.priorityFeePercentageMultiplier)
    .divn(100);

  const suggestedMaxPriorityFeePerGas = BN.max(
    adjustedPriorityFee,
    settings.minSuggestedMaxPriorityFeePerGas,
  );
  const suggestedMaxFeePerGas = adjustedBaseFee.add(
    suggestedMaxPriorityFeePerGas,
  );
}

```

Figure 4.5: Figure 4.3's *catch* branch ending flow, repeated for each priority level.

```

const SETTINGS_BY_PRIORITY_LEVEL = {
  low: {
    percentile: 10 as Percentile,
    baseFeePercentageMultiplier: new BN(110),
    priorityFeePercentageMultiplier: new BN(94),
    minSuggestedMaxPriorityFeePerGas: new BN(1_000_000_000),
    estimatedWaitTimes: {
      minWaitTimeEstimate: 15_000,
      maxWaitTimeEstimate: 30_000,
    },
  },
  medium: {
    percentile: 20 as Percentile,
    baseFeePercentageMultiplier: new BN(120),
    priorityFeePercentageMultiplier: new BN(97),
    minSuggestedMaxPriorityFeePerGas: new BN(1_500_000_000),
    estimatedWaitTimes: {
      minWaitTimeEstimate: 15_000,
      maxWaitTimeEstimate: 45_000,
    },
  },
  high: {
    percentile: 30 as Percentile,
    baseFeePercentageMultiplier: new BN(125),
    priorityFeePercentageMultiplier: new BN(98),
    minSuggestedMaxPriorityFeePerGas: new BN(2_000_000_000),
    estimatedWaitTimes: {
      minWaitTimeEstimate: 15_000,
      maxWaitTimeEstimate: 60_000,
    },
  },
};

```

Figure 4.6: Figure 4.5 additional.

In case of type-0 transaction (hence legacy, Figure 4.4) an HTTPS request is performed towards the URL specified as parameter (*fetchLegacyGasPriceEstimates*). The URL value, once again, is up to the live setting of an environment variable, opening the way to two possible assignments for it:

- <https://gas.api.cx.metamask.io/networks/1/gasPrices>
- <https://gas.uat-api.cx.metamask.io/networks/1/gasPrices>

One could simply copy and paste the given URLs to a web browser to see what *Metamask* is suggesting in real-time to those users involved in currency transfer operations over the Ethereum Mainnet network.

In case any problem occurs while dealing with one of the two snippets of code just discussed (Figure 4.3 or Figure 4.4), gas fee parameters are filled with the result coming from *eth_gasPrice*⁶ RPC.

⁶For details: https://getblock.io/docs/eth/json-rpc/eth_eth_gasprice/

A visualization of what results from the retrieved logic follows:

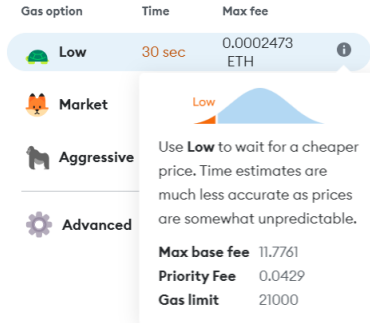


Figure 4.7: Metamask frontend gas fee suggestions for the low priority level along a type-2 transaction.

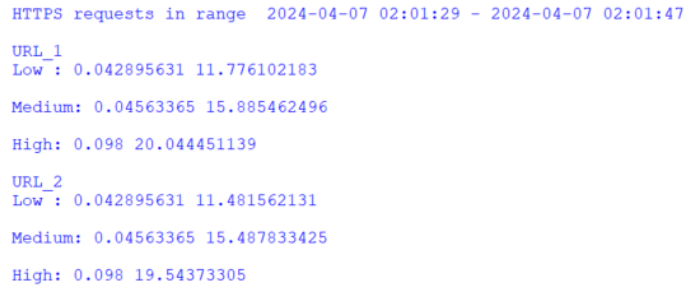


Figure 4.8: Collection of Metamask gas fee suggestions from the two URLs inherent to a type-2 transaction.

- Figure 4.7 shows gas fee suggestions made by *Metamask* wallet at sending ETH time. Among the three priority levels present, values associated with low priority level (*Max base fee* and *Priority Fee*) are highlighted.
- Figure 4.8 shows the results of the HTTPS requests (API calls) made for retrieving *Metamask* gas fee suggestions for type-2 transactions according to the source code. Results have been collected through a python script.

Looking at Figure 4.7 and Figure 4.8, one can notice the match between parameters suggested for the low priority level at frontend side (with *maxPriorityFee* parameter rounded) and the data obtained by performing the HTTPS request towards URL_1 (*https://gas.api.cx.metamask.io/networks/1/suggestedGasFees*).

For the purpose of the project, type-0 transaction gas fee suggestions are not needed (Section 4.2 for details): source code related to their computation will be skipped for the next wallets.

4.1.2 TRUST WALLET

Trust Wallet is a software wallet available as both a web extension (more limited) and a mobile application (for *Android* and *iOS*). It counts over 10 million downloads on the *Android* platform alone, showing its spread usage. If *Metamask* supports cryptocurrencies on the *Ethereum*, *Arbitrum*, *Binance Smart Chain*, *Optimism*, *Polygon*, and *Avalanche* networks, *Trust Wallet* supports all of these networks, plus dozens more. Features offered by *Metamask* (Figure 4.1)

are offered by *Trust Wallet* as well. At the frontend level, along with an ETH transfer, *Trust Wallet* prompts the user with one only gas fee suggestion level, specifying for it the *maxFeePerGas* and *maxPriorityFee* (miner's tip). By inspecting the application's source code⁷ and with the help of the *Trust Wallet Support Team*, it has been possible identifying the RPCs made in order to obtain those values then associated to the aforementioned parameters. The methods in question are *eth_feeHistory* and *eth_getBlockByNumber*. The first has been already discussed in Subsection 4.1.1, but a deeper discussion reveals to be necessary in this new app-related scenario.

- *eth_feeHistory* is used for the *maxPriorityFee* value assignment along an ETH transfer activity of a type-2 transaction. The method invocation occurs according to the following parameters:

```
{
  "id":1,
  "jsonrpc":"2.0",
  "method":"eth_feeHistory",
  "params":[
    10,
    "latest",
    [5]
  ]
}
```

Figure 4.9: *Trust Wallet*'s RPC parameters.

meaning that for each of the last ten blocks mined over the blockchain, the 5th percentile will be extracted (see Subsection 4.1.1 for clarifications). According to the RPC results, an array containing the *maxPriorityFee* corresponding to the 5th for each block has been built. The computation of the median of the obtained array corresponds to the *maxPriorityFee* (miner tip).

- *eth_getBlockByNumber*⁸ RPC is performed according to the structure in Figure 4.10.

⁷For details: <https://github.com/trustwallet>

⁸For details: <https://docs.alchemy.com/reference/eth-getblockbynumber>

```

eth_getBlockByNumber = {
  "jsonrpc": "2.0",
  "method": "eth_getBlockByNumber",
  "params": [
    "latest",
    True
  ],
  "id": 1
}

```

Figure 4.10: Trust Wallet's RPC parameters.

It will return informations about the last block mined into the Ethereum blockchain, in particular its `baseFeePerGas`.

Once these RPCs are performed (they could be emulated in *Python*, *Javascript*, *Postman* and more) and their results are retrieved, the `maxFeePerGas` parameter is obtained by increasing the retrieved `baseFeePerGas` of the 20%, then adding to the computed amount the miner's tip (`maxPriorityFee`). It translates into the formula

$$\text{MaxFeePerGas} = (\text{BaseFeePerGas} \times 1.2) + \text{MaxPriorityFee}$$

Figure 4.11 and Figure 4.12 show the perfect match between frontend proposed and script collected gas fees parameters.

Figure 4.11: Trust Wallet frontend gas fee suggestions for a type-2 transaction.

```

round ongoing
maxPriorityFee= 0.001
baseFeePerGas= 12.036584746
maxFeePerGas= 14.444901695

```

Figure 4.12: Collection of Trust Wallet gas fee suggestion for a type-2 transaction through a *python* script emulating the logic of Trust Wallet.

4.1.3 SHAPESHIFT WALLET

ShapeShift Wallet is a software wallet available as mobile application (for *Android* and *iOS*) and web application⁹. It counts over 500k downloads on the Android platform alone. At the frontend level, along with an ETH transfer, *ShapeShift Wallet* prompts the user with three different gas fee suggestion levels: *slow*, *average* and *fast*. As well as in *Metamask* (Subsection 4.1.1), the higher the level (from *slow* to *fast*) the higher the gas fee suggestion in terms of *MaxFeePerGas* and *MaxPriorityFee*. The wallet does not offer the possibility for the user to customize the gas fees. The source code of the *ShapeShift Wallet*¹⁰ present on GitHub has been analyzed. From it, the API necessary for the gas fee suggestions retrieval has been found:

<https://api.ethereum.shapeshift.com/api/v1/gas/fees>

Results coming from the API have the following form:

```
@Example<GasFees>({
  gasPrice: '77125288868',
  baseFeePerGas: '77654025212',
  maxPriorityFeePerGas: '94000001',
  slow: {
    gasPrice: '77109280451',
    maxFeePerGas: '77744243213',
    maxPriorityFeePerGas: '90218001',
  },
  average: {
    gasPrice: '78637140239',
    maxFeePerGas: '79158075213',
    maxPriorityFeePerGas: '1504050001',
  },
  fast: {
    gasPrice: '85079071846',
    maxFeePerGas: '89883761218',
    maxPriorityFeePerGas: '12229736006',
  },
})
```

Figure 4.13: Example of results coming from the *ShapeShift Wallet* API.

In Figure 4.13, for each priority level three parameters are indicated: *gasPrice*, *maxFeePerGas*, *maxPriorityFeePerGas*. For each priority level, the wallet will suggest as official *maxFeePerGas* for the type-2 transaction the output of $\max\{\text{gasPrice}, \text{maxFeePerGas}\}$, with the official *maxPriorityFeePerGas* for the type-2 transaction equal to the one retrieved by the API itself.

⁹For having access to it: <https://app.shapeshift.com>

¹⁰For details: <https://github.com/shapeshift>

At the frontend level, what a user will see when involved in a transfer operation is:

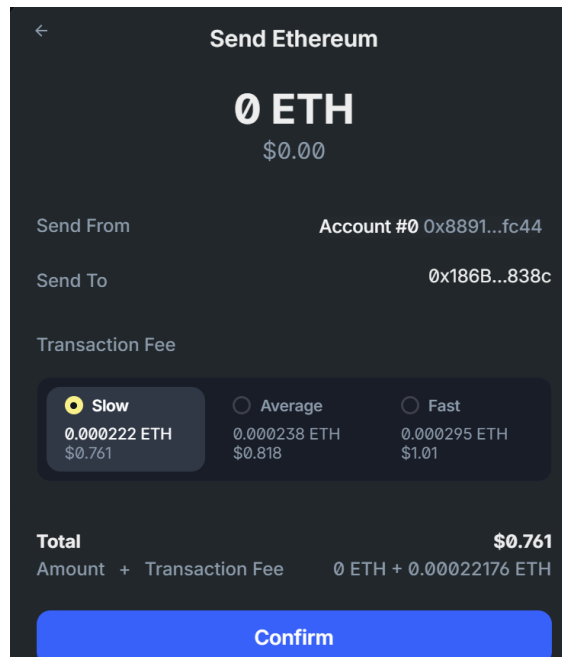


Figure 4.14: ShapeShift Wallet gas fee suggestions at the frontend level along a transfer operation.

with the gas fee-related values matching with those retrieved through a Python script built following the discussed logic:

```
SHAPESHIFT WALLET:  
SLOW : 0.000197001 10.560022863 hence transaction fee= 0.000222ETH  
STANDARD: 0.69707696 11.336838995 hence transaction fee= 0.000238ETH  
FAST: 3.082756627 14.026908415 hence transaction fee= 0.000295ETH
```

Figure 4.15: ShapeShift Wallet gas fee suggestions retrieved through a Python script.

4.1.4 RAINBOW

Rainbow is software wallet available as both mobile application (for *Android* and *iOS*) and web extension (e.g., *Chrome*, *Edge*, *Safari*, *Firefox*¹¹). It counts over 100k downloads on the *Android* platform alone. At the frontend level, along with an *ETH* transfer, *Rainbow* prompts the user with three different gas fee suggestion levels: *normal*, *fast* and *urgent*. The higher

¹¹For download: <https://rainbow.me/download>

the level, the higher the gas fee suggestion in terms of *MaxFeePerGas* and *MaxPriorityFee*. The wallet offers the possibility for the user to customize the gas fees. The source code of the *Rainbow* wallet¹² present on GitHub has been analyzed, identifying the packages path *rainbow/src/handlers/gasFees.ts*¹³ as the one containing a *typescript* file defining the URL to be reached for obtaining gas fee-related informations.

```

1  import { Network } from '@/helpers';
2  import { RainbowFetchClient } from '../rainbow-fetch';
3
4  const rainbowMeteorologyApi = new RainbowFetchClient({
5    baseURL: 'https://metadata.p.rainbow.me',
6    headers: {
7      'Accept': 'application/json',
8      'Content-Type': 'application/json',
9    },
10   timeout: 30000, // 30 secs
11 });
12
13 export const rainbowMeteorologyGetData = (network: Network) => rainbowMeteorologyApi.get(`/meteorology/v1/gas/${network}`, {});

```

Figure 4.16: *Rainbow*'s source code snippet with the reference to the API involved in the gas fee suggestions.

In particular, according to Figure 4.16, the URL to be reached in case of the *Ethereum Mainnet* network corresponds to:

https://metadata.p.rainbow.me/meteorology/v1/gas/mainnet

The informations retrievable from the addressed URL have the following form:

```

export interface RainbowMeteorologyData {
  data: {
    currentBaseFee: string;
    baseFeeSuggestion: string;
    baseFeeTrend: number;
    blocksToConfirmationByPriorityFee: BlocksToConfirmationByPriorityFee;
    blocksToConfirmationByBaseFee: BlocksToConfirmationByBaseFee;
    maxPriorityFeeSuggestions: MaxPriorityFeeSuggestions;
    secondsPerNewBlock: number;
  };
  meta: {
    blockNumber: number;
    provider: string;
  };
}

```

Figure 4.17: Output form of the Figure 4.16's API call.

¹²For details: <https://github.com/rainbow-me/rainbow>

¹³For details: <https://github.com/rainbow-me/rainbow/blob/77ef889186c89039e5152ec12b44fade8a1cfa44/src/handlers/gasFees.ts>

Among the resulting data structured according to Figure 4.17, those taken into consideration by the *Rainbow Wallet* for the three priority levels gas fee suggestions are:

- *baseFeeSuggestion*: the expected *baseFeePerGas* of the future next mined block;
- *maxPriorityFeeSuggestions*: an array of suggestions for the *maxPriorityFee* parameter for each of the proposed priority level.

Once data of interest are extracted, the *maxFeePerGas* associated with each proposed priority level is retrieved by incrementing the extracted *baseFeeSuggestion* of some percentages, according to what is reported in Figure 4.18.

```
const getBaseFeeMultiplier = (speed: string) => {  
  switch (speed) {  
    case URGENT:  
      return 1.1;  
    case FAST:  
      return 1.05;  
    case NORMAL:  
    default:  
      return 1;  
  }  
};
```

Figure 4.18: Priority levels multipliers.

According to Figure 4.18:

- *maxFeePerGas* for the *normal* level is equal to *baseFeeSuggestion*;
- *maxFeePerGas* for the *fast* level is obtained incrementing *baseFeeSuggestion* of its 5%;
- *maxFeePerGas* for the *urgent* level is obtained incrementing *baseFeeSuggestion* of its 10%.

The final *maxFeePerGas* suggested to the user for each priority level is finally modeled according to the nearest integer of the value obtained after the multiplication occurs. Figure 4.19, Figure 4.20 and Figure 4.21 show what a user sees at the frontend for each priority level.

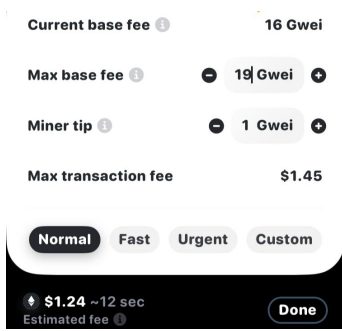


Figure 4.19: Rainbow wallet suggestions for the *normal* level.

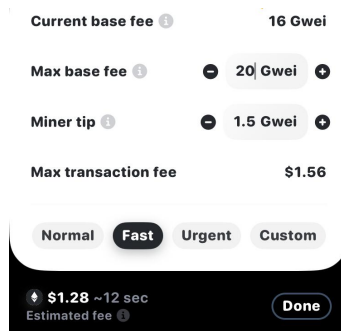


Figure 4.20: Rainbow wallet suggestions for the *fast* level.

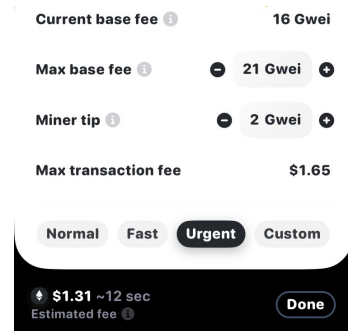


Figure 4.21: Rainbow wallet suggestions for the *urgent* level.

By comparing suggestions received from the frontend with a collection of the suggested fees coming from an emulation of the explained steps, one can notice perfect matches in terms of *maxFeePerGas* and *maxPriorityFee*.

```
RAINBOW WALLET:
1.0 19.160437986, hence 19
1.5 20.118459885300002, hence 20
2.0 21.076481784600002, hence 21
```

Figure 4.22: Rainbow gas fee predictions retrieved through a Python script. For the *maxFeePerGas* parameter, the picture shows both the float value and its rounding to the nearest integer.

4.1.5 ONEKEY

OneKey is a wallet available as mobile application (for *Android* and *iOS*), desktop application (for *macOS*, *Windows* and *Linux*¹⁴), web extension (for *Chrome*, *Edge* and *Brave*¹⁵) and hardware device. It counts over 50k downloads on the *Android* platform alone. At the frontend level, along with an *ETH* transfer, *Onekey* prompts the user with three different gas fee suggestion levels: *low*, *normal* and *high*. The higher the level (from *low* to *high*) the higher the gas fee suggestion in terms of *MaxFeePerGas* and *MaxPriorityFee*. The wallet offers the possibility for the user to customize the gas fees. The source code of the *OneKey* wallet¹⁶ present on *GitHub*

¹⁴For download: <https://onekey.so/download/>

¹⁵For download: <https://onekey.so/download/?client=browserExtension>

¹⁶For details: <https://github.com/OneKeyHQ>

has been analyzed, finding out that the developers rely on the *Blocknative*¹⁷ platform for the gas fee suggestion task. *Blocknative* is a real-time observability platform that offers several services, including a gas prediction tool. The tool makes its predictions based on real-time data coming from the *mempool* and predictive machine learning-based models. The combination of the two, allows the platform to predict the next block's minimum gas price as close as possible, avoiding overspend. Since different users have different degrees of urgency for getting transactions into the next block, the platform provides a range of confidence levels for next-block inclusion as well[25]:

- If a user needs a high probability of being confirmed within the next block at the expense of spending extra gas, he can use the 99% probability prediction.
- If a user does not mind if the transaction takes 2-3 blocks to be confirmed if it saves some gas, he can use the 70% probability prediction.

The two probability percentages are the extremes of a wider range of five: 99%, 95%, 90%, 80%, 70%. The smaller the probability, the smaller the associated gas fee predictions in terms of *MaxFeePerGas* and *MaxPriorityFee*. The HTTPS request necessary to retrieve the suggestions related to all the probability predictions corresponds to:

<https://api.blocknative.com/gasprices/blockprices>

A comparison of what a user sees at the frontend level by using the *OneKey* wallet and what has been retrieved from the *Blocknative* platform follows.

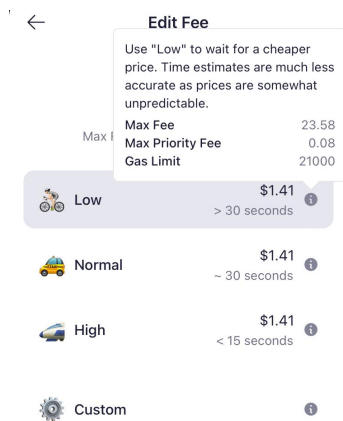


Figure 4.23: OneKey wallet suggestions for the low level.

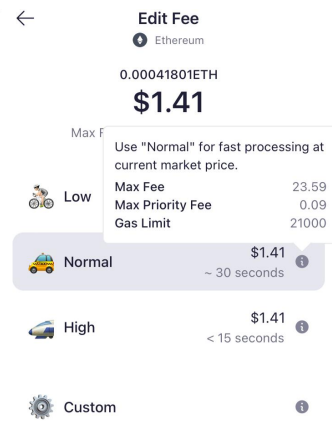


Figure 4.24: OneKey wallet suggestions for the normal level.

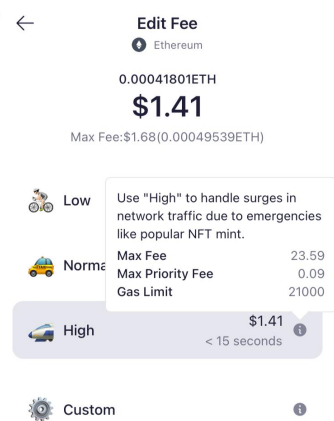


Figure 4.25: OneKey wallet suggestions for the high level.

¹⁷For details: <https://www.blocknative.com/gas-estimator>

Figure 4.23, Figure 4.24 and Figure 4.25 show the gas fees suggested by the *OneKey* wallet at ETH transfer time for each priority level presented. By comparing images' content with what is retrieved by the *Blocknative* platform (Figure 4.26) through the HTTPS request, parameters show to match.

```
ONE KEY WALLET:  
  
{0.1, 23.6}  
  
{0.09, 23.59}  
  
{0.09, 23.59}  
  
{0.08, 23.58}  
  
{0.07, 23.57}
```

Figure 4.26: Blocknative gas fee predictions for ranges of confidence in descending order (from 90% to 70%).

4.1.6 UNSTOPPABLE WALLET

Unstoppable Wallet is a software wallet available as mobile application for *Android* and *iOS*. It counts over 50k downloads on the Android platform alone. At the frontend level, along with an ETH transfer, *Unstoppable Wallet* prompts the transaction fee the transfer action will be subject to. The wallet offers the possibility for the user to customize the gas fees. The source code of the *Unstoppable Wallet*¹⁸ present on GitHub has been analyzed. According to it, the main point of the gas fee suggestion made by the wallet relies on the *eth_feeHistory* RPC, called with the following parameters:

¹⁸For details: <https://github.com/horizontalsystems>

```

{
  "id": 1,
  "jsonrpc": "2.0",
  "method": "eth_feeHistory",
  "params": [
    "10",
    "latest",
    [50]
  ]
}

```

Figure 4.27: Unstoppable Wallet RPC parameters.

According to Figure 4.27, informations concerning the last 10 blocks mined over the *Ethereum* blockchain are retrieved, with attention pointed over that *priorityFee* corresponding to the 50th percentile for each block (Subsection 4.1.1 for details). Results of the RPC are then handled according to the following code snippet, localized in the path of the packages *unstoppable – wallet–android/app/src/main/java/io/hizontalsystems/bankwallet/modules/evmfee/eip1559/Eip1559GasPriceService.kt*:

```

private fun handle(feeHistory: FeeHistory) {
    val recommendedBaseFee = max(recommendedBaseFee(feeHistory), minBaseFee ?: 0)
    currentBaseFee = recommendedBaseFee

    val recommendedPriorityFee = max(recommendedPriorityFee(feeHistory), minPriorityFee ?: 0)
    currentPriorityFee = recommendedPriorityFee

    val newRecommendGasPrice = GasPrice.Eip1559(recommendedBaseFee + recommendedPriorityFee, recommendedPriorityFee)

    recommendedGasPrice = newRecommendGasPrice

    if (recommendedGasPriceSelected) {
        state = validatedGasPriceInfoState(newRecommendGasPrice)
    } else {
        state.dataOrNull?.let {
            state = validatedGasPriceInfoState(it.gasPrice)
        }
    }
}

```

Figure 4.28: Figure 4.27's results handling.

According to what showed in Figure 4.28:

- *maxFeePerGas* parameter (*newRecommendedGasPrice* in the snippet) is computed as the *max* between two values (one of which is to be considered as zero), summed up to the computed *maxPriorityFee* (*recommendedPriorityFee* in the snippet). In order to

retrieve the arguments of the *max* function, the code of the function called inside it is needed.

```
private fun recommendedBaseFee(feeHistory: FeeHistory): Long {
    val lastNRecommendedBaseFeesList = feeHistory.baseFeePerGas.takeLast(lastNRecommendedBaseFees)
    return java.util.Collections.max(lastNRecommendedBaseFeesList)
}
```

Figure 4.29: Definition of *recommendedBaseFee* function in Figure 4.28.

According to Figure 4.29, one of the argument of the *max* function is retrieved by taking the *baseFeePerGas* information of the last two blocks coming from the *eth_feeHistory* RPC and choosing the highest one between them.

- *maxPriorityFee* parameter (*newRecommendedPriorityFee* in the snippet) is computed as the max between two values (one of which is to be considered as zero), with the first argument delivered as result of the function *recommendedPriorityFee*.

```
private fun recommendedPriorityFee(feeHistory: FeeHistory): Long {
    var priorityFeesSum = 0L
    var priorityFeesCount = 0
    feeHistory.reward.forEach { priorityFeeArray ->
        priorityFeeArray.firstOrNull()?.let { priorityFee ->
            priorityFeesSum += priorityFee
            priorityFeesCount += 1
        }
    }
    return if (priorityFeesCount > 0)
        priorityFeesSum / priorityFeesCount
    else
        0
}
```

Figure 4.30: Definition of *recommendedPriorityFee* function in Figure 4.28.

According to Figure 4.30, the first argument of the *max* function is retrieved by summing up the percentiles coming from each of the last 10 blocks as result of the *eth_feeHistory* RPC and dividing the value obtained by the number of blocks (hence, 10).

At the frontend level, what a user will see when involved in transfer activity is:

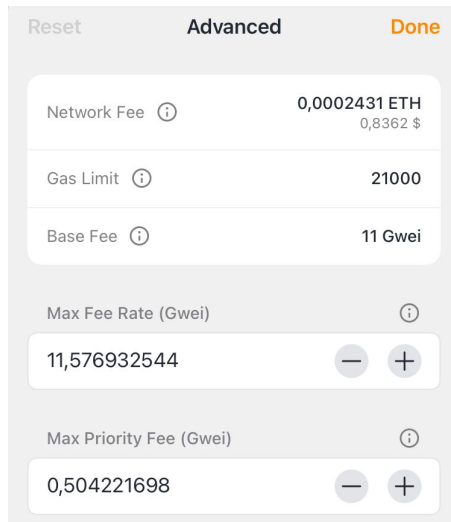


Figure 4.31: *Unstoppable Wallet* gas fee suggestions at frontend level.

whose gas fee parameters match with what is obtained by building a python script following the described logic:

```
UNSTOPPABLE WALLET:
0.504221698 11.576932544
```

Figure 4.32: *Unstoppable Wallet* gas fee suggestions retrieved through a python script.

4.2 TORNADO CASH ANALYSIS

As seen in Subsection 2.3.2, a withdrawal from a TC smart contract can be triggered in two different ways:

- By involving a relayer, without any need for the user to connect his wallet to the platform;
- Without involving a relayer, with the need for the user to connect his wallet to the platform.

In both cases, Tornado Cash makes its own gas fee suggestions (prompted as *network suggestion*), corresponding to the default one. In order to identify those transactions related to the TC platform whose assigned fees have no chance to come by any user wallet since proposed

by TC itself, the dApp open source code¹⁹ has been analyzed as well. The starting point of the analysis have been the *deposit* and *withdrawal* functions, both present in the *.sol* file in the packages path *tornado-core/contracts/Tornado.sol*. Both the functions implementation, along an ETH transfer (deposit or withdrawal), call the *generateTransaction*²⁰ function, with a code portion as follow:

```
async function generateTransaction(to, encodedData, value = 0) {
  const nonce = await web3.eth.getTransactionCount(senderAccount);
  let gasPrice = await fetchGasPrice();
  let gasLimit;
  ...
  function txoptions() {
    // Generate EIP-1559 transaction
    if (netId == 1) {
      return {
        to           : to,
        value        : value,
        nonce        : nonce,
        maxFeePerGas : gasPrice,
        maxPriorityFeePerGas : web3.utils.toHex(web3.utils.toWei('3', 'gwei')),
        gas          : gasLimit,
        data         : encodedData
      }
    }
    ...
  }
}
```

Figure 4.33: Portion of code related to the Tornado Cash's *generateTransaction* function.

According to Figure 4.33, the *maxFeePerGas* parameter for a TC deposit or withdrawal transaction is obtained by a call towards another function (*fetchGasPrice*), while *maxPriorityFeePerGas* is set as a constant, equal to the 3 value. The implementation of the *fetchGasPrice* function follows:

¹⁹For details: <https://github.com/tornadocash>

²⁰For details: <https://github.com/tornadocash/tornado-cli/blob/378ddf8b8b92a4924037d7b64a94dbfd5a7dd6e8/cli.js>

```

async function fetchGasPrice() {
  try {
    const options = {
      chainId: netId
    }
    // Bump fees for Ethereum network
    if (netId == 1) {
      const oracle = new GasPriceOracle(options);
      const gas = await oracle.gasPrices();
      return gasPricesETH(gas.instant);
    } else if (netId == 5 || isTestRPC) {
      const web3GasPrice = await web3.eth.getGasPrice();
      return web3GasPrice;
    } else {
      const oracle = new GasPriceOracle(options);
      const gas = await oracle.gasPrices();
      return gasPrices(gas.instant);
    }
  } catch (err) {
    throw new Error(`Method fetchGasPrice has error ${err.message}`);
  }
}

```

Figure 4.34: Portion of code related to the Tornado Cash's *fetchGasPrice* function.

From Figure 4.34, one can see that the task of the *maxFeePerGas* computation is delegated to another function (*gasPrices*), in particular the function is invoked over an object belonging to the external library *gas-price-oracle*²¹. After obtaining the outcome from the entity within the external library, the outcome's *instant* attribute is further processed through the *gasPricesETH* function, whose code follows:

```

function gasPricesETH(value = 80) {
  const tenPercent = (Number(value) * 5) / 100;
  const max = Math.max(tenPercent, 3);
  const bumped = Math.floor(Number(value) + max);
  return toHex(toWei(bumped.toString(), 'gwei'));
}

```

Figure 4.35: Portion of code related to the Tornado Cash's *gasPriceETH* function.

²¹For details: <https://github.com/peppersec/gas-price-oracle/tree/4861f36c56a12c8618141adcc55028d976fad7cd>

The code belonging to the external library has been dug in to understand what the input of the *gasPricesETH* function is. The external library function the attention has been put over is *gasPrices*, according to Figure 4.34. The function code has been divided into three code snippets for its presentation.

```
public async gasPrices(fallbackGasPrices?: GasPrice, shouldGetMedian = true): Promise<GasPrice> {
  if (!this.lastGasPrice) {
    this.lastGasPrice = fallbackGasPrices || this.configuration.fallbackGasPrices
  }

  const cacheKey = this.LEGACY_KEY(this.configuration.chainId)
  const cachedFees = await this.cache.get(cacheKey)

  if (cachedFees) {
    return cachedFees
  }

  if (Object.keys(this.offChainOracles).length > 0) {
    try {
      this.lastGasPrice = await this.fetchGasPricesOffChain(shouldGetMedian)
      if (this.configuration.shouldCache) {
        await this.cache.set(cacheKey, this.lastGasPrice)
      }
      return this.lastGasPrice
    } catch (e) {
      console.error('Failed to fetch gas prices from offchain oracles...')
    }
  }
}
```

Figure 4.36: Code snippet 1 of the *gasPrice* function in *gas-price-oracle* external library.

Figure 4.36 shows the first code snippet related to the *gasPrices* function. According to this path, the gas fee suggestions are retrieved by considering the outputs of some off-chain oracles (Section 2.2 for details), here corresponding to:

```
const ethgasstation: OffChainOracle = {
  name: 'ethgasstation',
  url: 'https://ethgasstation.info/json/ethgasAPI.json',
  instantPropertyName: 'fastest',
  fastPropertyName: 'fast',
  standardPropertyName: 'average',
  lowPropertyName: 'safeLow',
  denominator: 10,
  additionalDataProperty: null,
}
```

Figure 4.37: Oracle 1 taken into account by snippet in Figure 4.36.

```
const etherchain: OffChainOracle = {
  name: 'etherchain',
  url: 'https://etherchain.org/api/gasnow',
  instantPropertyName: 'rapid',
  fastPropertyName: 'fast',
  standardPropertyName: 'standard',
  lowPropertyName: 'slow',
  denominator: 1e9,
  additionalDataProperty: 'data',
}
```

Figure 4.38: Oracle 2 taken into account by snippet in Figure 4.36.

Oracle in Figure 4.37 is down from July 1st 2023²², so results are retrieved from Oracle in Figure 4.38 only. In particular, the API to be reached for the Oracle gas fee suggestion has been updated to :

<https://beaconcha.in/api/v1/execution/gasnow>

```
if (Object.keys(this.onChainOracles).length > 0) {
  try {
    const fastGas = await this.fetchGasPricesOnChain()

    this.lastGasPrice = LegacyGasPriceOracle.getCategorize(fastGas)
    if (this.configuration.shouldCache) {
      await this.cache.set(cacheKey, this.lastGasPrice)
    }
    return this.lastGasPrice
  } catch (e) {
    console.error('Failed to fetch gas prices from onchain oracles...')
  }
}
```

Figure 4.39: Code snippet 2 of the *gasPrice* function in *gas-price-oracle* external library.

Figure 4.39 shows the second code snippet related to the *gasPrices* function. According to this path, the gas fee suggestions are retrieved by considering the outputs of some on-chain oracles, here corresponding only to:

```
const chainlink: OnChainOracle = {
  name: 'chainlink',
  callData: '0x50d25bcd',
  contract: '0x169E633A2D1E6c10dD91238Ba11c4A708dfEF37C',
  denominator: '1000000000',
}
```

Figure 4.40: Oracle taken into account by snippet in Figure 4.39.

According to Figure 4.40, a RPC towards the specified smart contract address is performed, in particular the aim is that of triggering the smart contract's function identified by the value

²²For details: <https://ethgasstation.info/>

in *callData* parameter. Following up with Figure 4.39, the on-chain oracle output is then separately multiplied with different constants, in order to create several priority levels for the suggestion. Multipliers involved are those reported in Figure 4.41.

```
const MULTIPLIERS = {
  instant: 1.3,
  fast: 1.2,
  standard: 1.1,
  low: 1,
}
```

Figure 4.41: Multipliers for different priority levels generation valid for both Figure 4.39 and Figure 4.42 snippets.

GasPrice implementation ends up with the third reported snippet:

```
try {
  const fastGas = await this.fetchGasPriceFromRpc()

  this.lastGasPrice = LegacyGasPriceOracle.getCategorize(fastGas)
  if (this.configuration.shouldCache) {
    await this.cache.set(cacheKey, this.lastGasPrice)
  }
  return this.lastGasPrice
} catch (e) {
  console.error('Failed to fetch gas prices from default RPC. Last known gas will be returned')
}
return LegacyGasPriceOracle.normalize(this.lastGasPrice)
}
```

Figure 4.42: Code snippet 3 of the *gasPrice* function in *gas-price-oracle* external library.

Following the functions calling flow of Figure 4.42 leads to the *eth_gasPrice* RPC for the gas fee suggestions. Its output is subject to the multipliers in Figure 4.41 for the generation of the different priority levels.

As said, Figure 4.36, Figure 4.39 and Figure 4.42 are the three snippets composing the *gasPrice* function called in Figure 4.34. One of their outputs will be the one given as input to the function shown in Figure 4.35, where it will be summed up to the max between the 5% of the value itself and the 3 value, corresponding to the fixed *maxPriorityFee* parameter set by default by the TC platform, as shown in Figure 4.33.

Despite the made portrait, **empirically** it has been seen that some TC gas fee suggestions follow the following formula instead:

$$eth_gasPrice + 3 - 0.01$$

where:

- *eth_gasPrice* is the value resulting from the homonymous RPC;
- the 3 value corresponds to the amount assigned to *maxPriorityFee* parameter;
- the 0.01 value has been retrieved empirically.

An example of a TC transaction (withdrawal) involving a relay and matching the TC suggested fee retrieved according to the just showed empirically retrieved formula follows:

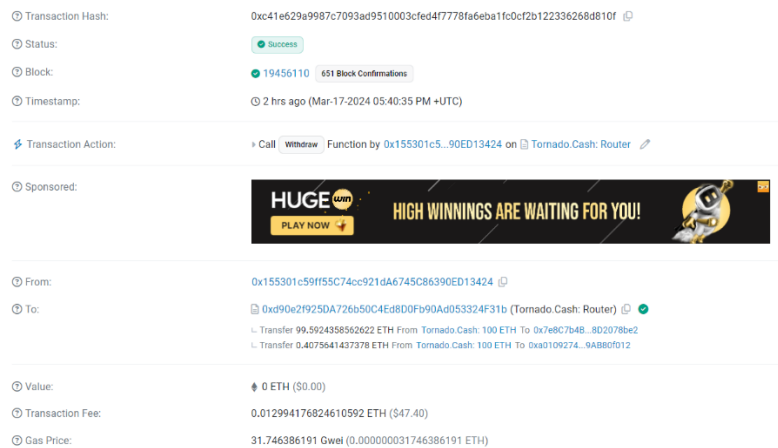


Figure 4.43: Example of a withdrawal involving a relay and the 100 ETH smart contract.

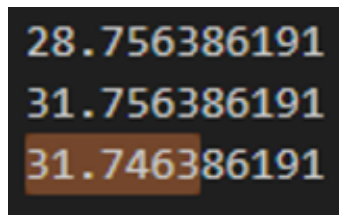


Figure 4.44: Tornado Cash gas fee suggestions according to the above formula.

In particular:

- The transaction present in Figure 4.43 is a legacy one (type-0), with a *gasPrice* parameter as specified. Generally, withdrawals involving a relayer are type-0 transactions, whereas withdrawals not involving a relayer (hence, those of interest for the project), are type-2 transactions. This is why the analysis of wallets has been centered around type-2 transactions.
- The three values present in Figure 4.44 correspond, respectively, to the *eth_gasPrice* RPC's result, $eth_gasPrice + 3$, $eth_gasPrice + 3 - 0.01$. The last of the three values is the one following the empirically retrieved formula and matching the TC transaction's *gasPrice* present in Figure 4.43 as well.

4.3 ADDITIONAL WORK

The analysis seen until now is all related to the *Ethereum mainnet* network (*chainId=1*), chosen because of its high usage within the TC platform. The high usage information comes from the visualization of the related smart contracts activity, hence the daily occurrences of deposits/withdrawals towards/from one of the *ETH* currency-related pools.

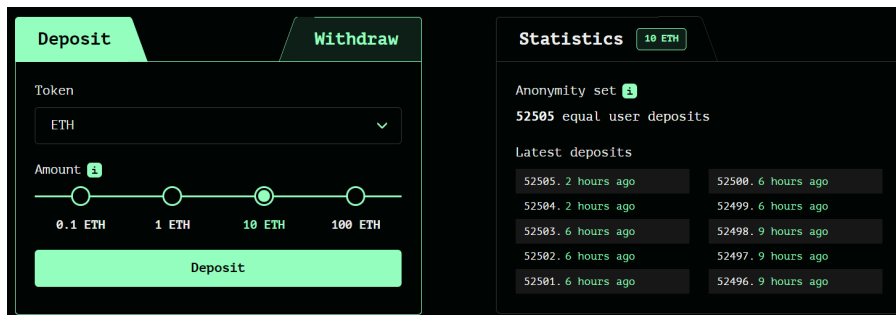


Figure 4.45: Example of TC activity concerning its 10 ETH pool in *Ethereum Mainnet* network.

Polygon network is second to *Ethereum Mainnet* in terms of its usage within the TC platform, with *MATIC* as its native currency.

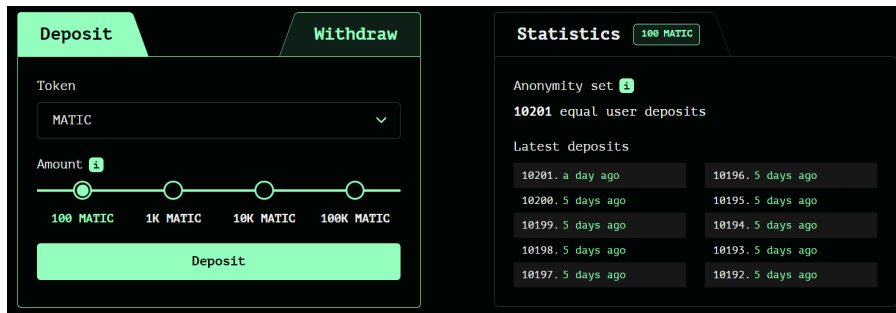


Figure 4.46: Example of TC activity concerning its 100 MATIC pool in Polygon network.

With the aim of examining how wallets retrieve those values then suggested as gas fees to users involved in a MATIC transfer employing the *Polygon* network, the open source code of several *WalletConnect* compatible software wallets has been analyzed for this circumstance as well. Wallets analyzed are some of those already discussed for the *Ethereum* scenario, plus an additional one. This conveys in the following analyzed wallets for the *Polygon* case: *Metamask*, *OneKey*, *Rainbow*, *ShapeShift Wallet* and *Rabby Wallet*

METAMASK As well as in the *Ethereum* case, suggestions for MATIC transfer occurring through Metamask come from two possible APIs, according to the real-time setting of an environment variable (see Subsection 4.1.1 for details). In this case, the two possible APIs correspond to:

<https://gas.api.cx.metamask.io/networks/137/suggestedGasFees>

<https://gas.uat-api.cx.metamask.io/networks/137/suggestedGasFees>

From both the APIs, gas fee suggestions in terms of *maxFeePerGas* and *maxPriorityFeePerGas* for the *low*, *medium* and *high* priority levels are retrievable.

ONEKEY Gas fee suggestions are made by the *OneKey* wallet by relying on the *Blocknative* platform (see Subsection 4.1.5 for details). In *Polygon* case, suggestions coming from the platform are retrievable through the API reachable by the following URL:

<https://api.blocknative.com/gasprices/blockprices?chainid=137>

The API returns the *maxFeePerGas* and *maxPriorityFeePerGas* for several levels of confidence, to be intended as priority levels.

RAINBOW Gas fee suggestions are made by the *Rainbow* wallet by relying on the following:

<https://metadata.p.rainbow.me/meteorology/v1/gas/polygon>

The output consists of three gas prices, each for a different priority level (see Subsection 4.1.4 for details). The *Rainbow* wallet multiplies each of the retrieved gas prices with the 1.05 multiplier and suggests to the user the ceiling of that.

SHAPESHIFT WALLET Gas fee suggestions are made by the *ShapeShift* wallet by relying on the following API:

<https://api.polygon.shapeshift.com/api/v1/gas/fees>

The output consists of three gas fee suggestions (see Subsection 4.1.3 for details), one for each proposed priority level. In particular, each result's row is made of three parameters: *gasPrice*, *maxFeePerGas* and *maxPriorityFeePerGas*. According to these values, a gas fee suggestion made by the wallet for each priority level corresponds to:

$$\begin{aligned}\text{maxFeePerGas_official} &= \max(\text{maxFeePerGas}, \text{gasPrice}), \\ \text{maxPriorityFeePerGas_official} &= \text{maxPriorityFeePerGas}\end{aligned}$$

RABBY WALLET *Rabby Wallet* is a software wallet available as mobile application (for *Android* and *iOS*), web extension for *Chrome*²³ and desktop application. It counts over 10k downloads on the *Android* platform alone. At the frontend level, along with a *MATIC* transfer, *Rabby Wallet* prompts the user with three different gas fee suggestion levels: *standard*, *fast* and *instant*. The higher the level, the higher the gas fee suggestion. The wallet offers the possibility for the user to customize the gas fees. The source code of the *Rabby Wallet*²⁴ present on *GitHub* has been analyzed, resulting in the following API as the one used for the gas fee suggestions retrieval:

https://api.rabby.io/v1/wallet/gas_market?chain_id=matic

²³For download: <https://rabby.io/>

²⁴For details <https://github.com/RabbyHub/Rabby>

A comparison of what is seen by a user at the frontend level and what retrieved by the above API follows, showing a match between the two:

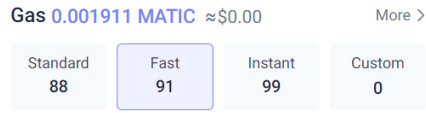


Figure 4.47: Rabby Wallet suggested fees at frontend level.

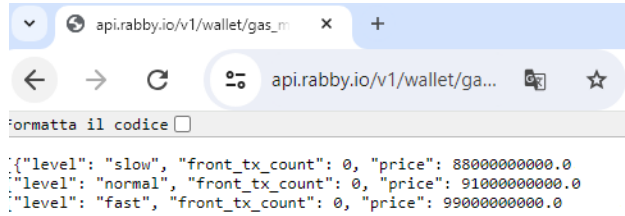


Figure 4.48: Rabby Wallet suggested fees in wei retrieved through the API.

5

Results

Each of the analyzed wallet in Chapter 4 follows its own approach for the gas fee suggestions task: it does not happen for two different wallets to propose the same suggestions. A gas fee suggestion is considered as associated with a single wallet according to a one-to-one relationship. A *Python* script (*python* version 3.12.1) has been written up to emulate the logic that each analyzed wallet exploits for the gas fee suggestions. The *python* script has been run over a remote server (a virtual machine reached through the *ssh* command) for a time window covering about one month: from 01/02/2024 to 09/04/2024. The *python* script execution results in a *.txt* file filled with a collection of the gas fee suggestions made by each wallet, in particular, the suggestions have been collected, on average, every 15 seconds. The short time slot between a collection and the next one is due to the dynamic nature of the *Ethereum* network, whose congestion changes second after second with gas fee suggestions following such a flow. Each single collection (a single emulation, at a specific time, of the gas fee suggested by each analyzed wallet) is to be intended as the one for a type-2 transaction (both the *maxPriorityFee* and *maxFeePerGas* parameters are retrieved) and is made up of the following data:

- The timestamp (+UTC) of the emulation occurrence. The information has been saved according to the *Universal Time Coordinated* (UTC) since it is the one transactions in Etherscan are recorded with.
- Collection of gas fee suggestions performed by the *Metamask* Wallet for all of the three proposed priority levels (*low, medium and high*). Suggestions are collected for both the potential APIs reached by the wallet (see Subsection 4.1.1 for details).

- Collection of gas fee suggestions performed by the *OneKey* Wallet for all of the proposed confidence intervals (99%, 95%, 90%, 80%, 70%, see Subsection 4.1.5 for details).
- Collection of gas fee suggestions performed by the *Rainbow* Wallet for all of the three proposed priority levels (*normal*, *fast* and *urgent*).
- Collection of gas fee suggestions performed by *Trust Wallet*.
- Collection of gas fee suggestions performed by the *ShapeShift* Wallet for all of the three proposed priority levels (*slow*, *standard* and *fast*).

An example of a one-shot collection made by the *Python* script follows, with the *maxPriorityFee* corresponding always to the first collected value and *maxFeePerGas* corresponding always to the second one:

```

HTTPS requests in range 2024-03-04 12:57:54 UTC - 2024-03-04 12:58:00 UTC
METAMASK WALLET:
Low : 0.00100298 70.173773954
Medium: 0.01979399266 94.753034808
High: 0.05363298332 119.34734364

Low : 0.00100298 70.173773954
Medium: 0.01979399266 94.753034808
High: 0.05363298332 119.34734364

ONE KEY WALLET:
{0.1, 89.63}
{0.09, 89.62}
{0.09, 89.62}
{0.08, 89.61}
{0.07, 89.6}

RAINBOW WALLET:
1.0 86.418931759
1.5 90.73987834695001
2.0 95.06082493490001

TRUST WALLET:
0.01135435000000001 85.345630361

UNSTOPPABLE WALLET:
0.924810872 74.083694902

SHAPESHIFT WALLET:
SLOW : 0.200241181 73.35912521
STANDARD: 4.355288649 77.514172678
FAST: 10.019666119 83.178550148

```

Figure 5.1: Example of gas fee suggestions collected at a specific UTC time by the built *Python* script.

Informations present in the *.txt* file collecting multiple elements following the format shown in Figure 5.1 have been compared with the gas fees associated with transactions present over the *Ethereum* blockchain. Transactions initiated within the *Ethereum* network have been explored using Etherscan. In particular, transactions of interest are:

- Transactions involved in a transfer of currencies over the whole Ethereum network;
- Transactions involved in interactions with one of the Tornado Cash smart contracts related to the ETH currency over the *Ethereum* network.

Transactions involved in a transfer of currencies over the whole *Ethereum* network have been analyzed with the purpose of revealing the potentiality of the used approach. The final aim of the project is to reveal if the heuristic concerning wallet fingerprints is a valid one for reducing the Tornado Cash pools' anonymity set, by checking the feasibility of associating made transactions with the wallet they come from. Once the association is made, withdrawals performed through a specific wallet are considered indistinguishable among deposits coming from the same wallet only. Analyzing the approach on a general scenario first (whole *Ethereum* network) and on the specific one afterward (*Tornado Cash*) works as a verification step: we make sure that the approach is valid in general, hence that in general it is possible to link a transaction involved in a cryptocurrency transfer with the wallet it has been initialized from. Once this is proven, it is possible to move on to the more specific scenario. The number of transactions implying a transfer action over the *Ethereum* blockchain in the time window going from 01/03/2024 to 09/04/2024 is very large. To prove the feasibility of the approach, a smaller time window has been taken into account: the one going from 09/04/2024 08:02:00 pm to 09/04/2024 11:28:47 pm, covering more than three hours. Transactions present over the *Ethereum* blockchain in the specified time window have been filtered according to those triggered by a transfer action, identified by the action id *0xa9059cbb*, present in the input field of the transaction itself. This has been done since the collected gas fees are those related to the occurrence of ETH transfers over the blockchain (transactions triggered by different actions could imply a different approach for the gas fee suggestions followed by the wallet itself). Transactions covering the specified time window have been identified as those present in blocks going from 19620335 to 19621362. A *javascript* file has been built up to query each block in the range to extract transactions triggered by a transfer action. This has been possible by exploiting the *Web3* library and a private *infura* key. An overview of the synthesized script follows:

```

const {Web3} = require('web3')
const fs = require('fs');
var url= 'https://mainnet.infura.io/v3/*****' // put your INFURA key in place of *
var web3 = new Web3(url)
const nameFileToSave= 'etherscanTRANSACTIONS.txt'

async function getTransactionsHashesInTimeInterval(startBlock, endBlock) {
  for (let blockNumber = startBlock; blockNumber <= endBlock; blockNumber++) {
    const block = await web3.eth.getBlock(blockNumber, true);
    if (block && block.transactions && block.transactions.length > 0) {
      block.transactions.forEach(transaction => {
        if( transaction.input.startsWith('0xa9059cbb')){
          var type=Number(transaction.type)
          if(type==2){
            var maxFeePerGas = web3.utils.fromWei(transaction.maxFeePerGas, 'gwei');
            var maxPriorityFeePerGas = web3.utils.fromWei(transaction.maxPriorityFeePerGas, 'gwei');
            fs.appendFile(nameFileToSave,transaction.hash.toString() + "\n" + maxFeePerGas.toString()+" " +
              "+ maxPriorityFeePerGas.toString()+ "\n"+ "\n", {}, (err) => {
              if (err) {
                console.error("Error:", err);
                return;
              }
            });
          }
        }
      });
    }
  }
}

async function main() {
  const startBlock =19621338
  const endBlock = 19621362
  console.log('START')
  await getTransactionsHashesInTimeInterval(startBlock, endBlock)
  console.log('END, check the created file!')
}
main();

```

Figure 5.2: Javascript code snippet for retrieving (and filter) transactions from the *Ethereum* blockchain.

Through the *javascript* code in Figure 5.2, 66948 transactions have been extracted from the blockchain, in particular their hash and related gas fees have been saved in a new *.txt* file. A *python* script has been built up to compare the gas fees attached to each of the extracted transactions with the collected suggestions, coming from the analyzed wallet in the same time window. The comparison resulted in 13203 full matches, where a full match is obtained when:

- *maxPriorityFee* and *maxFeePerGas* parameters attached to a transaction are equal to those suggested by an analyzed wallet;
- the gas fee suggestion collection has occurred a few seconds before the transaction validation (the validation time, hence the time between the transaction triggering and its validation over the blockchain, is taken into account).

Examples of full matches follow:

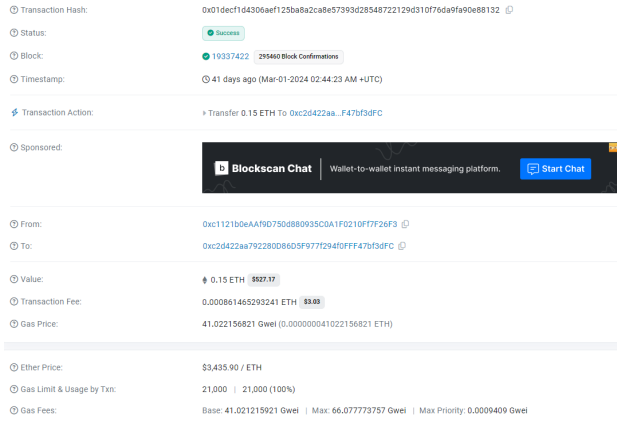


Figure 5.3: Example of a real transaction validated over the *Ethereum* Blockchain.

HTTPS requests in range 2024-03-01 02:43:53 UTC - 2024-03-01 02:44:01 UTC
 METAMASK WALLET:
 Low : 0.000884446 48.946686562
 Medium: 0.0009409 66.07773757
 High: 0.0009506 83.208814198

Low : 0.00088444882 45.912377626
 Medium: 0.0009409 61.981456689
 High: 0.05311392632 78.102652328

ONE KEY WALLET:
 {0.1, 53.55}
 {0.09, 53.54}
 {0.09, 53.54}
 {0.08, 53.53}
 {0.07, 53.52}

RAINBOW WALLET:
 1.0 54.232951315
 1.5 56.94459888075
 2.0 59.656246446500006

TRUST WALLET:
 0.00097 55.094761812

UNSTOPPABLE WALLET:
 0.221805631 46.133298088

SHAPESHIFT WALLET:
 SLOW : 0.000319913 46.755769169
 STANDARD: 0.891625649 47.899143244
 FAST: 3.685007884 53.015448742

Figure 5.4: Gas fee suggestion collections coming from the analyzed wallets for a timestamp consistent with the transaction in Figure 5.3.

Figure 5.3 and Figure 5.4 show a full match occurring between gas fee attached to a transaction over the *Ethereum* blockchain (in terms of *maxFeePerGas* and *MaxPriorityFee*) and the suggestions for the same values coming from the *Metamask* wallet.

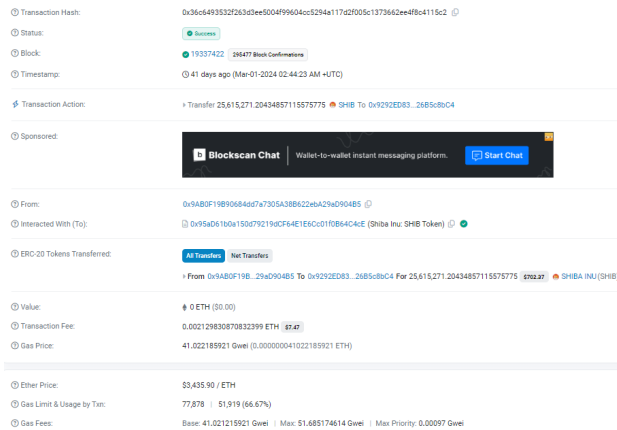


Figure 5.5: Example of a real transaction validated over the *Ethereum* Blockchain.

HTTPS requests in range 2024-03-01 02:44:09 UTC - 2024-03-01 02:44:15 UTC
 METAMASK WALLET:
 Low : 0.00088444882 43.071054961
 Medium: 0.0009409 58.145671092
 High: 0.05311392632 73.272403797

Low : 0.00088444882 43.071054961
 Medium: 0.0009409 58.145671092
 High: 0.05311392632 73.272403797

ONE KEY WALLET:
 {0.1, 53.85}
 {0.09, 53.84}
 {0.09, 53.84}
 {0.08, 53.83}
 {0.07, 53.82}

RAINBOW WALLET:
 1.0 49.538097867
 1.5 52.01500276035
 2.0 54.491907653700004

TRUST WALLET:
 0.00097 51.685174614

UNSTOPPABLE WALLET:
 0.366805631 43.436976143

SHAPESHIFT WALLET:
 SLOW : 0.000318545 46.5455143
 STANDARD: 0.841672694 47.668091271
 FAST: 5.929410219 53.008813981

Figure 5.6: Gas fee suggestion collections coming from the analyzed wallets for a timestamp consistent with the transaction in Figure 5.5.

Figure 5.5 and Figure 5.6 show a full match occurring between gas fee attached to a transac-

tion over the Ethereum blockchain (in terms of *maxFeePerGas* and *MaxPriorityFee*) and the suggestions for the same values coming from *Trust Wallet*.

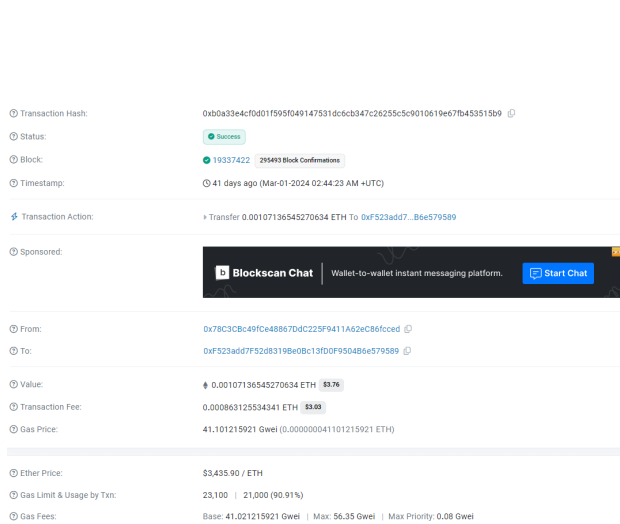


Figure 5.7: Example of a real transaction validated over the *Ethereum* Blockchain.

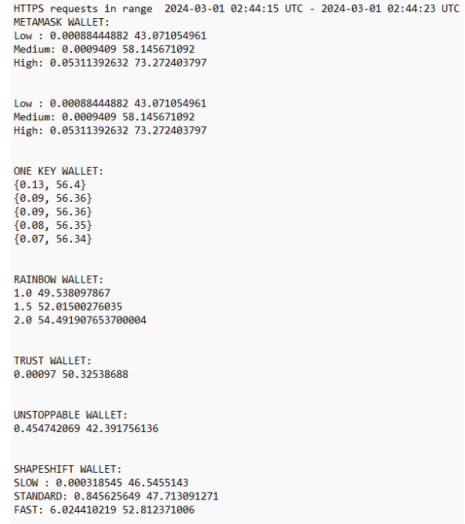


Figure 5.8: Gas fee suggestion collections coming from the analyzed wallets for a timestamp consistent with the transaction in Figure 5.7.

Figure 5.7 and Figure 5.8 show a full match occurring between gas fee attached to a transaction over the *Ethereum* blockchain (in terms of *maxFeePerGas* and *MaxPriorityFee*) and the suggestions for the same values coming from *OneKey Wallet*.

According to the results, 13203 transactions over 66248 present gas fee parameters matching with those collected, meaning that for each of the 13203 transactions, the wallet that transaction has been initiated from is now a public information. This not only serves as a support for the approach application over the more specific scenario of Tornado Cash, but represents a privacy concern over the *Ethereum* blockchain: information related to the wallet used by a user to perform a transaction is not a public one, but the followed approach allows to retrieve such information for some transactions. In particular, the 66248 - 13203 transactions not matching the collection are due to:

- Customized gas fees: if a user decides not to accept the wallet-suggested gas fees but to set fees by himself, the approach does not work.
- Other wallets: the collection is made up of the gas fee suggestions coming from the 6 analyzed wallets. Wallets usable over *Ethereum* are far away more than 6, implying that users employing a not analyzed wallet are undetected.

Given the support provided for the approach by testing it over transactions validated within the *Ethereum* network in general, the heuristic has been tested over transactions addressing Tornado Cash smart contracts (pools) as well. In particular, transactions interacting with one of the Tornado Cash pools existing over the *Ethereum Mainnet* network (0.1 ETH, 1 ETH, 10 ETH, 100 ETH) and involving the ETH cryptocurrency transfer are those interacting with the contract address

0xd90e2f925DA726b50C4Ed8D0Fb90Ad053324F31b

The address is that belonging to the *Tornado.Cash: Router*. Over a time window of about one month (from 01/03/2024 to 09/04/2024), 4579 transactions have occurred with respect to the contract address. Among the 4579 transactions:

- 2210 are deposits,
- 2088 are withdrawals involving a relayer,
- 281 are withdrawals not involving a relayer.

The total number of transactions involving the ETH currency on the *Ethereum* network through the Tornado Cash platform has been acquired through:

1. Manually downloading the *.csv* files in Etherscan once in the contract related page¹,
2. Extracting from each *.csv* file the hash of each present transaction through the usage of Excel, then appending it to a purposely generated *.txt* file,
3. Gaining informations about each transaction present in the generated *.txt* file using its related hash as starting point. A *javascript* file has been built up for the purpose.

The *javascript* file that extracts informations about Tornado Cash related transactions using their hash as starting point has the following logic, exploiting, once again, *web3* and *infura*: the script takes as input a *.txt* file containing transactions hash only. For each hash, considered a unique identifier for a transaction over the blockchain, the following informations are retrieved:

¹*Tornado.Cash: Router* in Etherscan: <https://etherscan.io/address/0xd90e2f925da726b50c4ed8d0fb90ad053324f31b>

- Gas fees attached to the transaction identified by the given hash. For the purpose, the type of each transaction has been extracted (type-0 or type-2) in order to know the number of gas-related parameters associated with the transaction itself;
- The nature of the transaction, hence if it is a deposit or a withdrawal. The information has been gained by considering the *id* of the action triggering the transaction, reported in the transaction input field together with other parameters. In this case, *0xb438689f* represents a withdrawal while *0x13d98d13* represents a deposit;
- In case of withdrawal transactions, the script tells apart those involving a relayer and those that do not. This is done by digging into the input field content of the transaction itself. In the case of a relayer involved, his address over the blockchain and the amount of currency to him dedicated is expressed inside the transaction input field, otherwise the parameters leave space for zeros value. This translated into checking if specific positions inside the input field of a withdrawal transaction are filled with zero values or not.

As seen in Subsection 2.3.2, when a relayer is involved in a withdrawal transaction there is no need for the user to connect his wallet to the Tornado Cash platform. This is because the relayer will pay the fees on his own and send the remaining amount to the recipient's address through an ETH transfer. The situation is different when a relayer is not involved in a withdrawal transaction: in this case the withdrawer has to pay the transaction fees on his own, making it necessary to connect his wallet to the Tornado Cash platform. The case of withdrawal transactions not involving a relayer is the one wallet fingerprints heuristic can have an effect on. This means that among the 4579 TC-related transactions that occurred in the \approx one-month time window, those where the wallet fingerprints heuristic could have any effect are 4579-2088, hence withdrawals involving a relayer are out of the heuristic coverage. This turns into 2491 transactions to be analyzed. A *python* script has been built up for the purpose of comparing the TC-related transactions of interest (2210 deposits and 281 withdrawals not involving a relayer) with the collected gas fee suggestions by the analyzed wallet. The script execution turned in:

- 91 full matches with deposits,
- 22 full matches with withdrawals not involving a relayer;
- 0 full matches with withdrawals involving a relayer, as expected (when a relayer is involved, the user does not connect his own wallet to the dApp).

All deposits full matches and withdrawals full matches come from the same wallet: *Meta-mask*. Examples of full matches for both a deposit and a withdrawal concerning the same Tornado Cash smart contract (pool) follow:

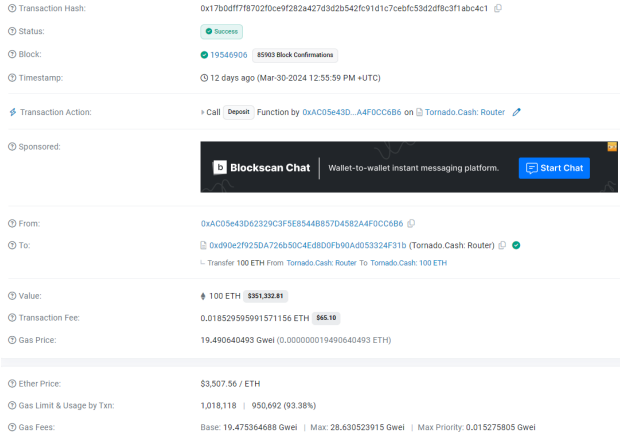


Figure 5.9: Example of a real deposit transaction validated over the Ethereum Blockchain. TC pool involved is 100 ETH.

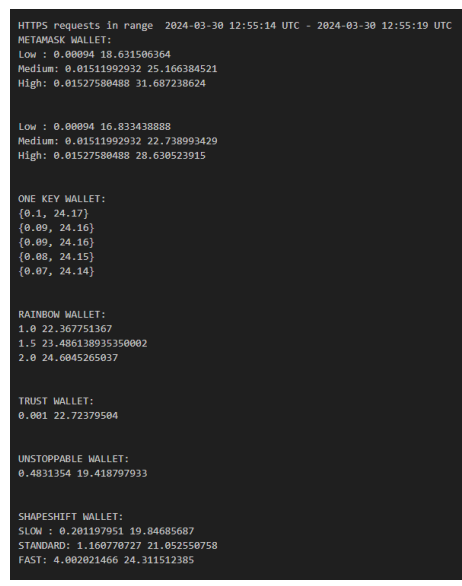


Figure 5.10: Full match in terms of gas fee suggestions coming from the Metamask wallet for a timestamp consistent with the transaction in Figure 5.9.

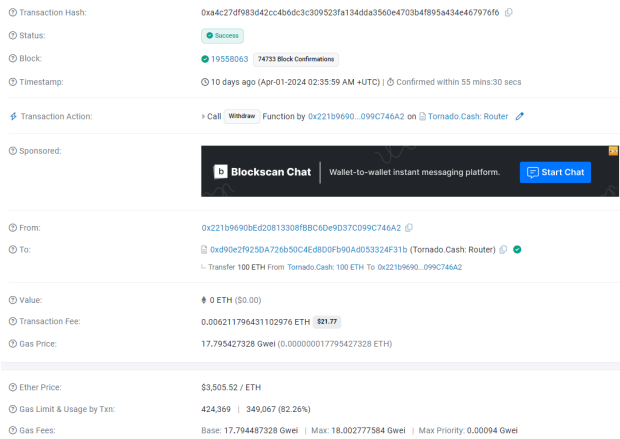


Figure 5.11: Example of a real withdrawal transaction validated over the Ethereum Blockchain. TC pool involved is 100 ETH.

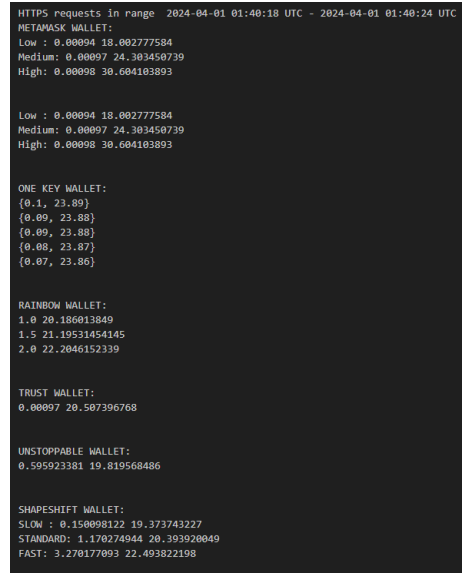


Figure 5.12: Full match in terms of gas fee suggestions coming from the Metamask wallet for a timestamp consistent with the transaction in Figure 5.11.

Being the deposit in Figure 5.9 and withdrawal in Figure 5.11 related to the same Tornado Cash pool and since the deposit has occurred before the withdrawal itself, the users related to these transactions are considered heuristically linkable according to the wallet fingerprints heuristic. The reason behind a so small number of full matches in the Tornado Cash scenario relies on:

- Possibility for the user to customize gas fees;
- Usage of other wallets with respect to those analyzed;
- Suggestions coming from the Tornado Cash platform itself.

In the Tornado Cash scenario, gas fees attached to a transaction can not only come as suggestions from a wallet or being customized, but the dApp itself makes its own suggestions. In particular, according to the empirically retrieved formula for gas fee suggestions coming from Tornado Cash, in a time window covering the period going from 17/03/2024 05:28:55 pm to 09/04/2024 11:28:47 pm (a subset of the time window previously analyzed) it happens that:

- 2856 transactions are collected: 1454 deposits and 1402 withdrawals, 162 of which do not involve a relayer;
- Over the 1454 deposits, 140 are full matches with the output of the empirically retrieved formula concerning gas fee suggestions coming from Tornado Cash itself;
- Over 1402 withdrawals, both involving a relayer or not, 253 are full matches with the output of the empirically retrieved formula concerning gas fee suggestions coming from Tornado Cash itself. In particular, 11 full matches concern withdrawals without a relayer involved.

It has to be noticed that gas fee suggestions coming from the Tornado Cash platform have no dependency on the wallet used at transaction time, hence they hold for both withdrawals with and without a relayer involved. Examples of gas fees matching between Tornado Cash suggestions according to the empirically retrieved formula and transactions within Tornado Cash follow:

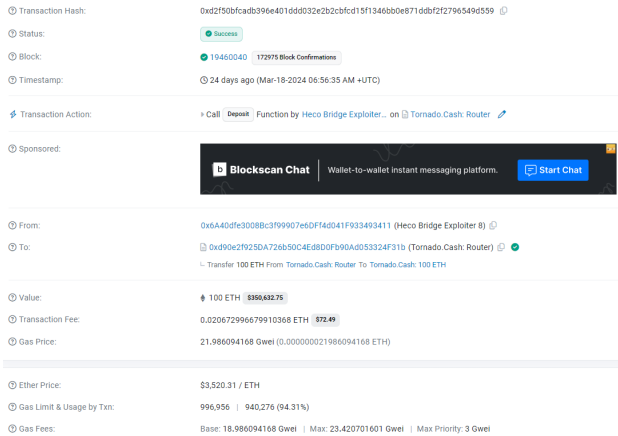


Figure 5.13: Example of a real Tornado Cash deposit transaction validated over the *Ethereum* Blockchain.

```

HTTPS requests in range 2024-03-18 06:55:45 - 2024-03-18 06:55:46
20.430701601000003
23.430701601000003
23.420701601

```

Figure 5.14: Gas fee suggestion collections coming from the TC platform according to the empirically retrieved formula for a timestamp consistent with the transaction in Figure 5.13.

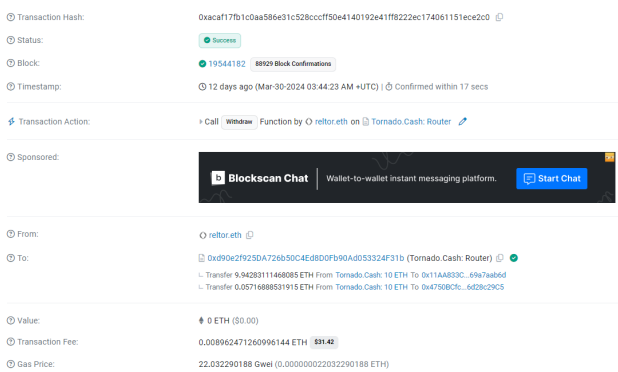


Figure 5.15: Example of a real Tornado Cash withdrawal transaction with relay involved validated over the *Ethereum* Blockchain.

```

HTTPS requests in range 2024-03-30 03:44:03 - 2024-03-30 03:44:03
19.042290188000003
22.042290188000003
22.032290188

```

Figure 5.16: Gas fee suggestion collections coming from the TC platform according to the empirically retrieved formula for a timestamp consistent with the transaction in Figure 5.15.

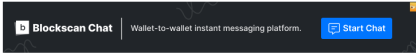
Transaction Hash: 0x298a1d348aa6f391b5625e686df335d728f85bd5c88566bde3757169bc221840

Status: Success

Block: 19458746 17424 Block Confirmations

Timestamp: 24 days ago (Mar-18-2024 02:35:11 AM +UTC)

Transaction Action: [Call](#) [Withdraw](#) Function by 0x72c97c69...F635DDE5F on [Tornado Cash: Router](#)

Sponsored:  [Start Chat](#)

From: 0x72c97c6976251c1307f5a193d954b51f635DDE5F

To: 0xd990c2f9250a726b50c4e8800f96a4d55324f31b (Tornado Cash: Router)

Transfer 1 ETH From Tornado.Cash: 1 ETH To 0x72c97c69...F635DDE5F

Value: 1 ETH (\$0.00)

Transaction Fee: 0.009801433796602875 ETH \$34.30

Gas Price: 28.079912325 Gwei (0.000000028079912325 ETH)

Ether Price: \$3,520.31 / ETH

Gas Limit & Usage by Txn: 550,000 | 349,055 (63.46%)

Gas Fees: Base: 25.628156844 Gwei | Max: 28.079912325 Gwei | Max Priority: 3 Gwei

Figure 5.17: Example of a real Tornado Cash withdrawal transaction with no relayer involved validated over the *Ethereum* Blockchain.

HTTPS requests in range 2024-03-18 02:34:46 - 2024-03-18 02:34:47

25.089912325

28.089912325

28.079912325

Figure 5.18: Gas fee suggestion collections coming from the TC platform according to the empirically retrieved formula for a timestamp consistent with the transaction in Figure 5.17.

6

Conclusion

Tornado Cash is a dApp retrofitting with privacy several networks, with Ethereum as the main one. The privacy added by the mixer over Ethereum has been challenged by several heuristics (see Chapter 3 for details), which analyze mainly the user behavior within the Tornado Cash platform. This project introduces and evaluates a novel heuristic that diverges from focusing solely on user interactions with the dApp. Instead, it targets the typical operational flow of Tornado Cash. Termed as *wallet fingerprints* [1][10], this heuristic aims to establish a connection between blockchain transactions and the originating wallet. Applying this concept within the Tornado Cash ecosystem: transactions, including withdrawals and deposits, can be categorized (clustered) based on the wallet they have been initialized from. A cluster of withdrawals is considered linkable to a cluster of deposits when they share the same originating wallet across both clusters. Analysis of the logic behind real-time gas fees suggested by several wallets has been necessary for the purpose. The wallet fingerprints approach has been tested in the *Ethereum* network generic scenario first for support reasons. Testing conducted in this general case has revealed that out of a collection of 66,948 transactions validated on the Ethereum blockchain, $\approx 20\%$ yields to full matches with the data obtained from the analyzed wallets. The ability to link a blockchain transaction to the originating wallet raises a privacy concern, as it involves retrieving non-public information. After achieving positive results on the Ethereum network as a whole, the wallet fingerprints heuristic has been further tested specifically on transactions involving Tornado Cash smart contracts (pools) on Ethereum (with deposit amounts of 0.1 ETH, 1 ETH, 10 ETH, 100 ETH). This testing has revealed that out of a subset of 2,491 eligi-

ble Tornado Cash-related transactions within a one-month timeframe (excluding withdrawals involving a relayer), $\approx 4,5\%$ results in full matches with respect to the collected wallet data. In particular, the full matches include 91 deposits and 22 withdrawals, with the implication that deposits and withdrawals initiated by the same wallet and interacting with the same Tornado Cash pool are considered heuristically linkable, provided that the deposit precedes the withdrawal in terms of timestamp. An analysis of the gas fee suggestions provided by the Tornado Cash platform itself has been conducted as well. This analysis has led to the empirical derivation of a formula representing one of the methods through which TC generates network suggestions. According to this empirically derived gas fee suggestions formula, out of 2,856 transactions occurring within TC's Ethereum-related pools, 343 ($\approx 12\%$) are full matches, indicating that their gas fee parameters are set according to the suggestions provided by the dApp. Transactions not covered by the approach are due to:

- Customized gas fee;
- Gas fee set by wallets not analyzed yet;
- Gas fee suggested by TC according to a logic different from the empirically retrieved one.

The proposed approach demonstrates its impact both in the Ethereum blockchain, with the rise of a privacy concern, and in TC, taking into account that in the last scenario only withdrawals not involving a relayer have any chance to be affected by the heuristic. Moreover, for the heuristic to be effective, a dynamic collection of gas fees suggested by each wallet must be created and regularly updated. Transactions occurring in a time window not covered by the collection can not be affected by the proposed wallet fingerprints heuristic. The fingerprints are comprised of gas fee suggestions collected in real-time from the following analyzed wallets: *Metamask*, *Trust Wallet*, *Rainbow*, *OneKey Wallet*, *ShapeShift Wallet* and *Unstoppable Wallet*.

The same wallet can support different blockchain networks (e.g., *Bitcoin*, *Ethereum*, *Arbitrum*, *Gnosis*, *Polygon*), providing for each of them different gas fee suggestions. Additional work has been dedicated to uncovering the gas fee suggestion logic employed by several analyzed wallets specifically for the Polygon network. The wallets analyzed for the purpose include *Metamask*, *OneKey*, *Rainbow*, *ShapeShift Wallet*, and *Rabby Wallet*. A collection of the gas fees suggested by each mentioned wallet for a time window of one month in the case of the Polygon network has been made as well. Future efforts could be put in:

- Testing the additional obtained collection in both the *Polygon* network and the Tornado Cash smart contracts to the network related, hence repeating the steps performed in this project but changing the main character. This could be done in order to reveal a privacy concern over the Polygon network, as well as revealing clusters of users employing Tornado Cash platform within Polygon, second to Ethereum in terms of Tornado Cash platform usage.
- Digging into the logic exploited by the already analyzed wallet concerning other blockchain networks, e.g., *Arbitrum*, *Gnosis*, *Avalanche*.
- Extending the wallets analysis in the *Ethereum* case to other open source wallets, e.g., *Torus Wallet*, *MEW Wallet*, *Guarda Wallet*, *MyCrypto Wallet*. This expansion would likely lead to an increase in the number of full matches observed in both the *Ethereum* general case and the Tornado Cash scenario.

Each of the proposed future directions represents a meaningful extension of the current project, contributing to its overall value and relevance.

Acknowledgments

I express my gratitude towards the Trust Wallet support team, because of their patience, attention and meticulousness in following up with me in details concerning the wallet source code.

References

- [1] M. Wu, W. McTighe, K. Wang, I. Seres, N. Bax, M. Puebla, M. Mendez, F. Carrone, T. Matthey, H. Obst Demaestri, M. Nicolini, and P. Fontana, “Tutela: An open-source tool for assessing user-privacy on ethereum and tornado cash,” 01 2022.
- [2] Y. Tang, C. Xu, C. Zhang, Y. Wu, and L. Zhu, “Analysis of address linkability in tornado cash on ethereum,” in *Cyber Security*, W. Lu, Y. Zhang, W. Wen, H. Yan, and C. Li, Eds. Singapore: Springer Nature Singapore, 2022, pp. 39–50.
- [3] H. Du, Z. Che, M. Shen, L. Zhu, and J. Hu, “Breaking the anonymity of ethereum mixing services using graph feature learning,” *IEEE Transactions on Information Forensics and Security*, vol. PP, pp. 1–1, 01 2023.
- [4] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” *Cryptography Mailing list at <https://metzdowd.com>*, 03 2009.
- [5] M. Dotan, A. Lotem, and M. Vald, “Haze: A compliant privacy mixer,” Cryptology ePrint Archive, Paper 2023/1152, 2023, <https://eprint.iacr.org/2023/1152>. [Online]. Available: <https://eprint.iacr.org/2023/1152>
- [6] Chainalysis. (2022) Tornado cash: Sanctions challenges. [Online]. Available: <https://www.chainalysis.com/blog/tornado-cash-sanctions-challenges/>
- [7] Z. Wang, S. Chaliasos, K. Qin, L. Zhou, L. Gao, P. Berrang, B. Livshits, and A. Gervais, “On how zero-knowledge proof blockchain mixers improve, and worsen user privacy,” 01 2022.
- [8] Z. Wang, X. Xiong, and W. J. Knottenbelt, “Blockchain transaction censorship: (in)secure and (in)efficient?” Cryptology ePrint Archive, Paper 2023/786, 2023, <https://eprint.iacr.org/2023/786>. [Online]. Available: <https://eprint.iacr.org/2023/786>
- [9] Y. Zhou, J. Wu, and S. Zhang, “Anonymity analysis of bitcoin, zcash and ethereum,” in 2021 *IEEE 2nd International Conference on Big Data, Artificial Intelligence and Internet of Things Engineering (ICBAIE)*, 2021, pp. 45–48.

- [10] F. Béres, I. A. Seres, A. A. Benczúr, and M. Quintyne-Collins, “Blockchain is watching you: Profiling and deanonymizing ethereum users,” in *2021 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*, 2021, pp. 69–78.
- [11] S. Werner, P. Pritz, and D. Perez, *Step on the Gas? A Better Approach for Recommending the Ethereum Gas Price*, 10 2020, pp. 161–177.
- [12] (2019) Pedersen hash. [Online]. Available: https://iden3-docs.readthedocs.io/en/latest/iden3_repos/research/publications/zkproof-standards-workshop-2/pedersen-hash/pedersen.html
- [13] R. S. Alexey Pertsev, Roman Semenov, “Tornado cash privacy solution,” December 2019, version 1.4.
- [14] J. Yang, S. Gao, G. Li, R. Song, and B. Xiao, “Reducing gas consumption of tornado cash and other smart contracts in ethereum,” in *2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2022, pp. 921–926.
- [15] T. Chen, H. Lu, T. Kunpittaya, and A. Luo, “A review of zk-snarks,” 02 2022.
- [16] M. Nadler and F. Schär, “Tornado cash and blockchain privacy: A primer for economists and policymakers,” *Review*, vol. 105, 01 2023.
- [17] S. Bistarelli, B. Montalvo, I. Mercanti, and F. Santini, *An E-Voting System Based on Tornado Cash*, 01 2023, pp. 120–135.
- [18] Tornado cash classic anonymity mining. [Online]. Available: <https://github.com/tornadocash/docs/blob/en/tornado-cash-classic/anonymity-mining.md>
- [19] H. Sun, N. Ruan, and H. Liu, “Ethereum analysis via node clustering,” in *International Conference on Network and System Security*, 2019. [Online]. Available: <https://api.semanticscholar.org/CorpusID:209325123>
- [20] T. Hu, X. Liu, T. Chen, X. Zhang, X. Huang, W. Niu, J. Lu, K. Zhou, and Y. Liu, “Transaction-based classification and detection approach for ethereum smart contract,” *Information Processing Management*, vol. 58, no. 2, p. 102462, 2021. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0306457320309547>

- [21] M. S. Bhargavi, S. Katti, M. Shilpa, V. Kulkarni, and S. Prasad, “Transactional data analytics for inferring behavioural traits in ethereum blockchain network,” 09 2020, pp. 485–490.
- [22] W. Chan and A. Olmsted, “Ethereum transaction graph analysis,” in *2017 12th International Conference for Internet Technology and Secured Transactions (ICITST)*, 2017, pp. 498–500.
- [23] S. Suratkar, M. Shirole, and S. Bhirud, “Cryptocurrency wallet: A review,” in *2020 4th International Conference on Computer, Communication and Signal Processing (ICCCSP)*, 2020, pp. 1–7.
- [24] MetaMask, “Metamask core repository,” <https://github.com/MetaMask/core>, Last access: 2024-02-14, fetchBlockFeeHistory.ts - Line 136.
- [25] Blocknative. (s.d.) Blocknative documentation. [Online]. Available: <https://docs.blocknative.com/>