Detecting Privileged Parties on Ethereum

Michael Fröwis and Rainer Böhme

Department of Computer Science, Universität Innsbruck, Austria

Abstract. The promise of smart contracts (computer programs running on a decentralized virtual computer) lies in the ability to execute agreements without the risk of interference by powerful intermediaries. However, in practice, many smart contracts reintroduce privileged parties on the application layer. They are programmed to enforce that certain functions can only be executed by the owners of defined accounts. We propose and validate a method to detect such privileged parties from binary smart contract code on the Ethereum platform. Our open-source implementation, Ethpector, can be used to verify claims about "zero-trust," reveal ownership structures, forensically analyze networks of virtual shell organizations, and may support auditors when testifying ownership of intangible assets on Ethereum held by conventional legal entities.

Keywords: Smart Contracts, Governance, Audits, Forensics, Ownership Pattern, Ethereum Virtual Machine, Symbolic Execution

1 Introduction

Privileged parties are entities that can unilaterally exercise control over a system on which ordinary users interact. Decentralized systems are often designed to minimize the influence of privileged parties in order to reduce the number of entities users have to trust. This design principle can make systems more robust to failures of individual components and arguably more trustworthy as a whole.

While, "zero-trust" transactions will probably remain a utopian vision [3,16], Ethereum today provides the technical means to avoid certain trust relationships. Yet, at the current state of Ethereum and its ecosystem, it is often unclear if services offered on the platform indeed require less trust that centralized alternatives. In many cases, trust relationships are merely less apparent to users, disguised in opaque program code, and renounced in marketing language.

It is instructive to illustrate this on a simplified technology stack as shown in Figure 1. A decentralized network of miners participates in a protocol which establishes consensus on the state of a replicated machine. This machine offers a programming interface for applications (i. e., smart contracts) to run on. While a decentralized platform is a *necessary* precondition for decentralized applications, it is *not sufficient*. Developers can re-introduce privileged parties on a higher layer by specifying that certain functions check the identity (i. e., Ethereum address) of the caller against constants or state variables before execution.

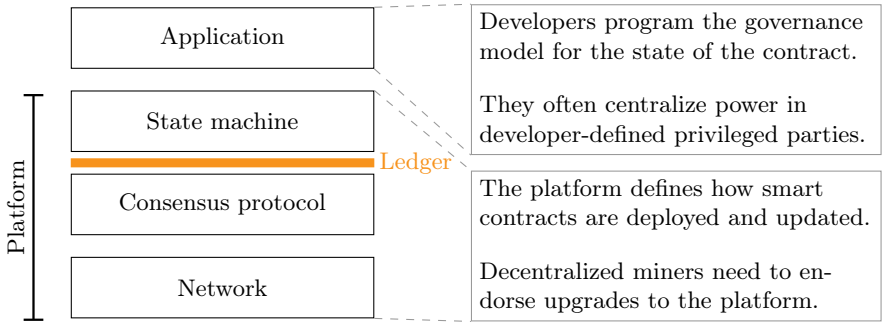

Fig. 1: Governance on different layers of the Ethereum technology stack.

To understand why privileged parties matter, consider the issues involved in updating smart contracts. Whenever code on Ethereum is prepared to be updatable (e. g., to be able to patch security vulnerabilities), there is some party in control of the update [13]. This party has exceptional control over whatever this application does. In fact, for the scope of the application, this party's power is comparable to that of an operator of a centralized system. For example, the party authorized to update the smart contract of a token system is at least as powerful as a conventional bank, the very type of intermediary cryptocurrency systems set out to remove. The party can freeze accounts, adjust balances, or shut down the entire token system. Even if there is no 'wild-card' option to replace the code, the deployed code may allow privileged parties to update parameters of the system, such as creating or destroying tokens, adjusting fees, or reassigning the privilege to other parties. From a users' perspective, this situation could be worse than when dealing with a regulated bank because there is little hope for legal redress. Besides legal and regulatory uncertainty, operators of token systems are barely accountable. They are often identified by nothing more than pseudonyms on social media or code sharing platforms. These examples highlight the need for a technical method to detect the privileged parties of any given smart contract before interacting with it.

Existing solutions to this problem are unsatisfactory. The canonical approach to identify all privileged parties (or verify the absence of them) is to review the source code of the smart contract. In a second step, one has to replicate the compile process to ensure that the deployed code indeed implements the functionality specified in the source code. Third, one has to inspect the transaction that deployed the smart contract to ensure that it cannot change in the future [10]. This laborious manual process requires deep technical understanding of the platform as well as of the application in question. Clearly, this approach does not scale. It also does not work for smart contracts whose source code is not publicly available.

Contribution: This paper presents `Ethpector`, a tool that facilitates the automatic extraction of privileged parties from smart contract binaries. We start with a motivating example in Section 2. Section 3 proposes our method to extract

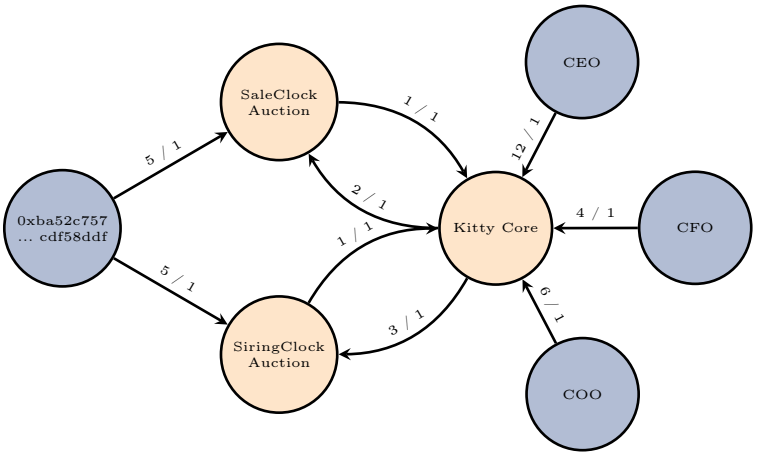

Fig. 2: The governance structure of the CryptoKitties contract. Orange nodes represent code accounts, gray nodes are externally owned accounts. The edge annotations mean: number of functions controlled / by n parties (one slot).

privileged parties, which we evaluate against a curated set of ground-truth data in Section 4. We discuss applications and limitations in Section 5. Section 6 discusses related work before Section 7 concludes. `Ethpector` is an open-source implementation of the proposed method. It can be used to replicate all results in this paper. We provide a brief description and feature list of the artifact in the appendix.

2 Motivating Example

Established in late 2017, CryptoKitties belong to the first digital collectibles on the Ethereum platform. They became precursors of the NFT hype. Figure 2 visualizes the governance structure of the CryptoKitties contract.[1] The addresses labeled CEO, COO, and CFO can execute the privileged function `pause` to halt all contract activity, which in effect freezes all kitties. Furthermore, the CEO is authorized to invoke `setGeneScienceAddress`, which sets a new reference to the account responsible for creating kitties. This address controls how unique new kittens are and thus may influence their valuation [17]. The power of these privileged parties stands in contrast to the claims made on the CryptoKitty website:[2]

[1] The UNESCO defines governance as: "...structures and processes that are designed to ensure accountability, transparency, responsiveness, rule of law, stability, equity and inclusiveness, empowerment, and broad-based participation."; See `http://www.ibe.unesco.org/en/geqaf/technical-notes/concept-governance`, Accessed: 14 June 2022

[2] `https://www.cryptokitties.co/about`, Accessed 18 Jan 2022

> "... each CryptoKitty is one-of-a-kind and 100 % owned by you. It cannot be replicated, taken away, or destroyed."

Meanwhile, users seem to happily trust the privileged parties of this contract, or might simply be unaware of this governance structure.

Our `Ethpector` tool, which generated the governance structure in Figure 2, is designed to extract such privileged parties automatically from a smart contract binary. This enables users to scrutinize claims and understand power relations before sending funds to a contract. Applied iteratively, our method allows to reveal ownership structures and track them through networks of "shell contracts", Ethereum's equivalent to shell companies that are commonly used to conceal wealth and decision power in the real world.

Yet another use case of `Ethpector` is when auditors certify virtual assets on a real-world entity's balance sheet. While holdings of ether and standard tokens can be verified with common techniques, there is no canonical way to testify ownership of – or other kinds of privileged access to – smart contracts. To the best of our knowledge, the method presented here is the first to tackle this problem in generality for arbitrary smart contracts.

3 Proposed Method

Smart contracts can implement privileged parties in many different ways. Our aim is to identify functions in EVM binaries where parts can only be executed by a privileged party. Moreover, we want to be able to identify the privileged party by its Ethereum address.

3.1 Dead End: Heuristic Pattern Matching

We first considered to heuristically extract information from the most common standardized interfaces. This approach as been used widely in the literature, for instance to analyze token flows from log entries of popular token standards [4,7,8,6,14,15,5]. We identified the *ownership pattern* as a common form to manage a single privileged party. As shown with code examples in Appendix C, this pattern keeps an owner address in the smart contract's state. This variable is typically initiated with the sender of the transaction that deploys the code. The pattern's interface supports an `getOwner` function, which can be called on the local node to identify the privileged party. Changes of ownership (i. e., of the privileged party) can be tracked by watching for calls to `changeOwner` or by observing `OwnerSet` logs emitted by the contract.

To evaluate the coverage of an approach solely relying on the ownership pattern, we classify all newly deployed smart contracts in a time window of two months in summer 2022 (891 170 contracts in total). We consider all smart contracts that export functions using Ethereum's Application Binary Interface (ABI). Among the contracts which do not export any function, we consider all

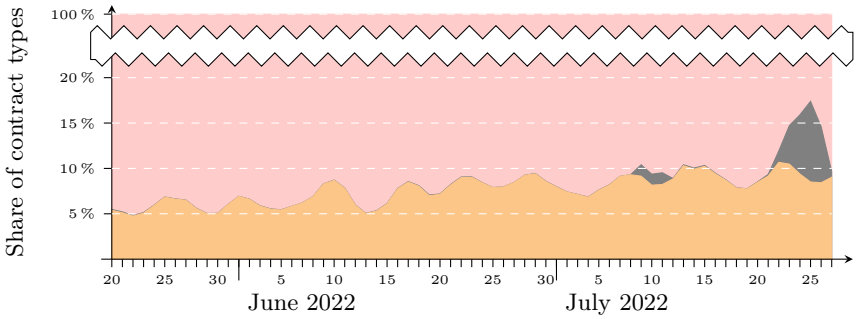

Fig. 3: Prevalence of the common ownership pattern in newly created smart contracts (orange) using a signature-based detector. Our baseline (red) are all contracts with known interfaces, such as ERC20. Gray contracts are likely multi-sig wallets, but cannot be analyzed with the proposed method.

deployments with known bytecode.[3] Those mainly include forwarder contracts used to create fresh deposit addresses and upgradable proxy contracts. Figure 3 shows the results over time using a 3-day rolling window. Observe that only 5–10 % of the relevant contracts implement the vanilla ownership pattern. The simple approach sketched in the previous paragraph would thus fail for at least 85 % of the cases where `Ethpector` can in principle extract some information. Also the CryptoKitties contract used as motivating example in Section 2 could not be analyzed with the ownership pattern heuristic. These numbers highlight the need for a more sophisticated approach using symbolic execution.

3.2 Symbolic Execution

Symbolic execution [11] is a common technique in program analysis, often used to find bugs or to generate test cases automatically. Symbolic execution tries to explore all possible execution paths through a program in a systematic manner. For that purpose, the input to the program is intentionally left unspecified (or *symbolic*). On each control flow decision, the symbolic execution engine tries to explore both branches. To avoid the exploration of unreachable paths, the engine uses a satisfiability modulo theories (SMT) solver to find program inputs that satisfy the path constraints leading to the branch. Paths for which no suitable input is found are skipped.

[3] For 36 % of the deployed contracts we cannot infer a contract type. They neither export functions nor belong to the our set of known bytecodes. The database of known interfaces and bytecodes is curated from public sources, e. g., `https://eips.ethereum.org/`, GitHub etc. A complete list of items can be found at `https://github.com/uibk-ethpector/ethpector/blob/main/src/ethpector/classify/classification.py`; function and event signatures are obtained from the 4-bytes directory and etherface.io.

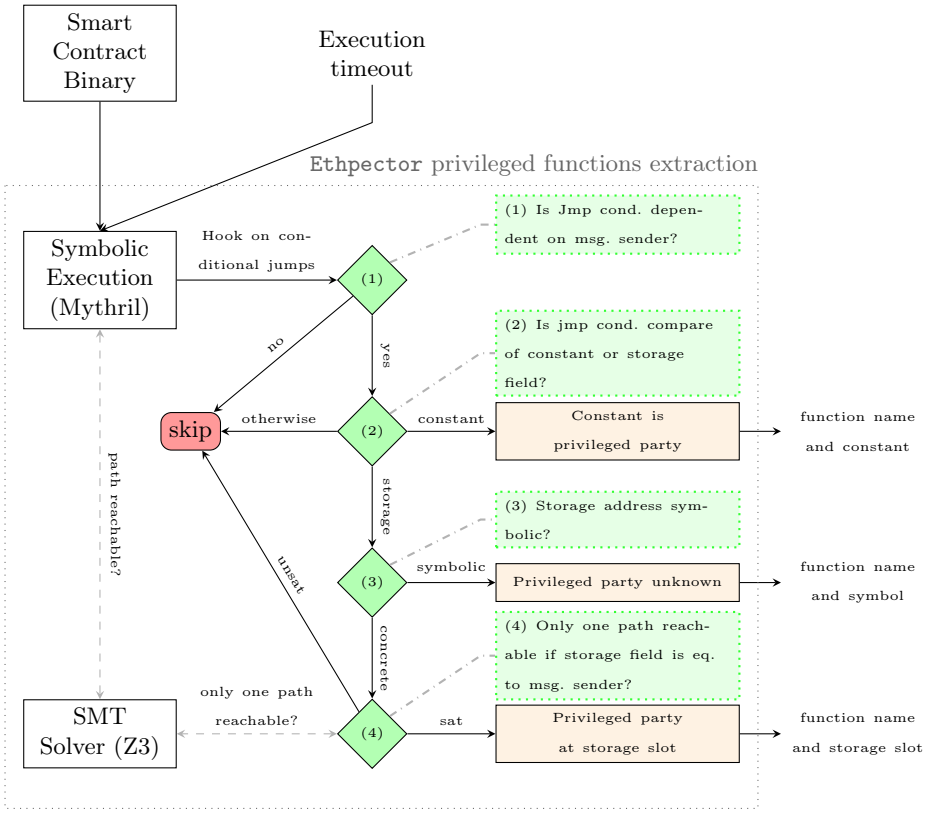

Fig. 4: Flow chart of `Ethpector's` privileged function detection.

For our purpose we are interested in control flow decisions within a function that depend on the caller, i. e., the sender of the message or transaction which invokes the function.[4] Moreover, we want to find control flow decisions where one of the branches can only be reached if the caller is a privileged party either (a) stored as constant in the binary or (b) read from a storage field. If we have found such a control flow decision, we want to extract the address of the privileged party. In case (a), we can read the encoded party directly from the binary. This means we go back to the `PUSH` instruction that placed the constant on the stack. In case (b), we need to extract the slot in permanent storage that holds the address. This storage slot is encoded in the path constraint and can be obtained by analyzing the structure of the path constraint. We use the path constraint as well as information we gain from tainting all values introduced by loads from permanent storage. We compare both values to avoid wrong extractions. Finally,

[4] In principle, one could also look for the origin, i. e., the party who signed the transaction. To the best of our knowledge, almost all authorization decisions on Ethereum are based on the message sender.

to ensure that the extracted data is correct, we use the SMT solver to verify that if we set the identified storage field to the current sender of the transaction then we in fact can only reach one of the branches.

Figure 4 shows the proposed binary analysis pipeline as a flow chart. Our `Ethpector` implementation uses Mythril[5] as symbolic execution engine, which in turn depends on Microsoft's Z3 SMT solver. We refer to Appendix B for more details on the concrete implementation and features of `Ethpector`. Observe that we can run into cases where we detect the existence of a privileged party, but fail to identify it. This can happen for multi-sig logic which conditions access on general boolean expressions. In all other cases, the analysis pipeline returns an address of the privileged party, or a storage slot. Given the extracted storage slot, we can look up the current value using the `getStorageAt` function of an Ethereum node. Repeating this look-up for different points in time on an archive node lets us track changes of the privileged party.

4 Validation

We validate the proposed method along several dimensions. First and foremost, we are interested in finding out if the method is able to correctly detect the existence of privileged parties. Then, for each detected privileged party, we are interested in whether it can be identified. Additional variables of interest are code coverage and execution time, which is not negligible given that symbolic execution tries to exhaustively explore all possible execution paths. Finally, we compare to the closest related work.

A main challenge in the validation is the lack of task-specific ground-truth data. The only reliable way to identify privileged parties independent of the proposed method is to carry out the manual code review described in Section 1. However, this depends on the availability of the source code. Since manual code review does not scale, we have to keep the size of the ground-truth data limited. This, in turn, means that the results become more sensitive to the sample selection. The naive approach to sample $x\,\%$ of all smart contracts deployed on Ethereum in the past year is prone to biases given that the overwhelming majority of deployments are proxy or forwarder contracts. Those are very short and generally easy to analyze. Therefore, such a sample would over-estimate the accuracy of the method while almost never testing its limits.

To overcome these issues, we compose a validation dataset of 41 non-trivial smart contracts by combining two sources. We identify 28 relevant smart contracts from the list of top gas consumers provided by EthGasStation.[6] Starting from the top, we take each smart contract for which Etherscan has source code. If the smart contract is invoked via a proxy pattern, we do not only analyze the proxy but try to include the contract on the address it resolves to. This applies to seven cases. Since the so-obtained dataset was still biased towards

[5] `https://github.com/ConsenSys/mythril`, Accessed: 07 June 2022

[6] `https://ethgasstation.info/json/gasguzz.json`, Accessed: 13 May 2022. The ranking aggregates gas use over 1500 blocks (roughly six hours).

Table 1: Extraction results compared to our ground-truth data.

	Name	F in Abi	TP F	FP F	FN F	TP O	FP O	FN O	Owners/Slots
0	0x: Coinbase Wallet Proxy	14	6	0	0	1	0	0	1 / 1
1	1inch v4: Router	23	4	0	0	1	0	0	1 / 1
2	BendDao: Bend Token	16	0	0	0	0	0	0	0 / 0
3	BendDao: Bend Token Proxy	6	6	0	0	1	0	0	1 / 1
4	BosonRouter	33	8	0	0	1	0	0	1 / 1
5	Center: USD Coin	52	12	0	0	2	0	0	2 / 5
6	Center: USD Coin Proxy	6	6	0	0	1	0	0	1 / 1
7	CryptoKitties	59	17	0	0	5	0	0	5 / 6
8	EMOBUDDIES	43	13	0	1	1	0	0	1 / 1
9	ENS: ETH Registrar Controller	22	5	0	0	1	0	0	1 / 1
10	FloatingCats	43	11	1	1	1	0	0	1 / 1
11	Gem: GemSwap 2	41	19	0	0	1	0	0	1 / 2
12	GnosisSafe Mastercopy 1.1.1	31	0	0	0	0	0	0	0 / 0
13	KaijuFrenz: KAIJUFRENZ Token	54	14	0	0	1	0	0	1 / 1
14	LooksRare: Exchange	23	7	0	0	0	1	1	1 / 1
15	Merkle distributor	19	7	0	0	1	0	0	1 / 1
16	Merkle distributor Proxy	6	6	0	0	1	0	0	1 / 1
17	MetaPunk: Minter	45	13	0	4	1	0	0	1 / 1
18	OpenSea: Registry	21	6	0	0	1	0	0	1 / 1
19	OpenSea: Wyvern Exchange v2	36	5	0	0	1	0	0	1 / 1
20	Polygon Matic: Bridge Proxy	7	3	0	0	1	0	0	1 / 1
21	Polygon Matic: Bridge RootChainManager	41	0	0	0	0	0	1	0 / 0
22	Proxy to 34cfac64 ... 83fe3f5f	1	0	0	0	0	0	0	0 / 0
23	ProxyAdmin ?	9	5	0	0	1	0	0	1 / 1
24	RaidParty: Game Party	28	0	0	0	0	0	0	0 / 0
25	RaidParty: Game Party Proxy	6	6	0	0	1	0	0	1 / 1
26	Saitama Inu: SAITAMA Token	24	5	1	1	1	0	0	1 / 1
27	Shiba Inu: SHIB Token	13	0	0	0	0	0	0	0 / 0
28	StrongBlock: Service Proxy	6	6	0	0	1	0	0	1 / 1
29	Strongblock: Service V19	102	32	0	0	1	0	0	1 / 7
30	Strongblock: Service V20	107	29	1	0	1	0	0	1 / 7
31	SushiSwap: Router	25	1	0	0	1	0	0	1 / 0
32	TRIBE X: TRIBEX Token	61	21	0	1	1	0	0	1 / 1
33	Tether: USDT Stablecoin	33	10	0	0	1	0	0	1 / 1
34	Tornado.Cash: Router	12	2	0	0	2	0	0	2 / 0
35	Uniswap V2: Router 2	25	1	0	0	1	0	0	1 / 0
36	Uniswap V3: Positions NFT	39	1	0	3	1	0	0	1 / 1
37	Uniswap V3: Router	18	1	0	0	1	0	0	1 / 0
38	Uniswap V3: Router 2	40	1	0	0	1	0	0	1 / 0
39	Wandernauts: WANDERNAUT Token	39	11	0	3	1	0	0	1 / 2
40	Wrapped Ether	12	0	0	0	0	0	0	0 / 0
	TOTAL:	1241	300	3	14	39	1	2	
	ACC, REC, F1:		0.99	0.96	0.97	0.98	0.95	0.96	

proxy contracts, we balance it by adding selected popular contracts of several diverse categories, such as multi-sig wallets, AdminProxy, and not to forget CryptoKitties. Table 2 in the appendix lists details for all elements of the validation dataset.

We generate ground-truth by downloading the source code for all contracts and manually extracting all functions that require authorization. This caused an effort of several working days by the Solidity expert on the research team. Additionally, we use Etherscan's read contract feature to collect the addresses set in permanent storage that are used for the authorization decisions. We use this manually curated data to evaluate the performance of our automated approach.

Table 1 reports the extraction performance of our method for each contract. We evaluate the capability to detect the existence of a privileged party on the level of functions. Column "F in Abi" lists the number of exported functions per contract. The following three columns count true positives, false positives, and false negatives, respectively. Overall, our method achieves 99 % accuracy and 96 % recall on a total of 1241 exported functions. Turning to the identification of privileged parties, the numbers are naturally smaller as few contracts handle

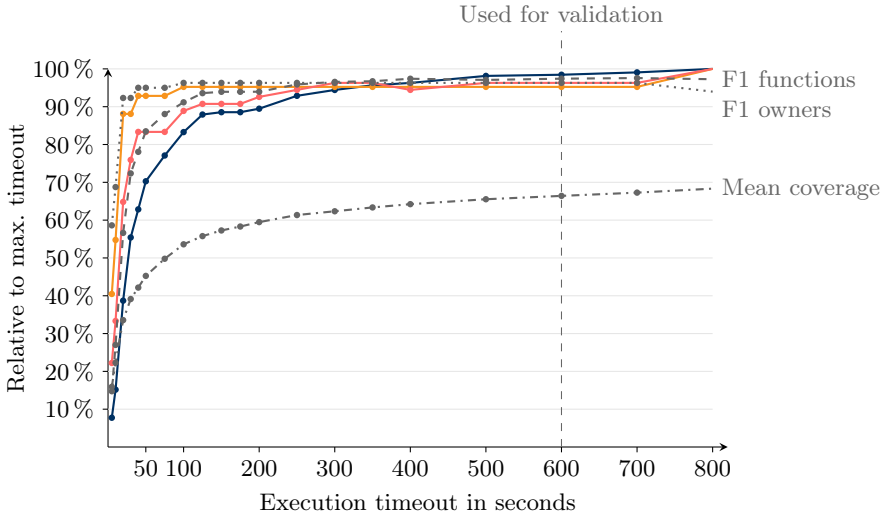

Fig. 5: Number of privileged functions (blue), owners (orange) and slots (red) found depending on the execution timeout of the symbolic execution.

more than one privileged party. We measure 98 % accuracy and 95 % recall across our validation dataset.

These results are obtained with a timeout of 600 seconds on a commodity AMD Ryzen 7 Pro[7] machine with 32 GB RAM. This runtime is not long enough to achieve complete code coverage except for very short contracts. To evaluate the sensitivity of the performance evaluation to the choice of the timeout, we repeat the extraction with varying timeout and report key indicators in Figure 5. Observe that the number of detected items plateaus after 300 seconds (and would allow a rough approximation after just 60 seconds). Likewise, the performance metrics (here: F1-scores) are largely insensitive to the timeout once it exceeds one minute. While this is still too slow to be attractive for realtime analysis of all deployed smart contracts, it is sufficiently fast for the exploration of governance structures in selected parts of the ecosystem. Note that the mean code coverage is far below 100 %. The nonetheless good performance takes advantage of the fact that authorization decisions typically appear early in the execution path and a breadth-first search strategy. This means the symbolic execution engine needs less time to reach the crucial instructions. Moreover, it is less likely that we miss the relevant branches since the path constraints remain lean and are unlikely to overwhelm the SMT solver. The mean code coverage of 65 % for the 600 second timeout hides tremendous heterogeneity, with individual coverages ranging between 20 and 100 %. As plotted in Figure 6 for each element in our validation set, the coverage is negatively correlated with the bytecode size, but can still vary by a factor of two between smart contracts of the same size.

[7] 5850U at 1.90–4.40 GHz, 8 cores, 16 threads, and 16 MB cache.

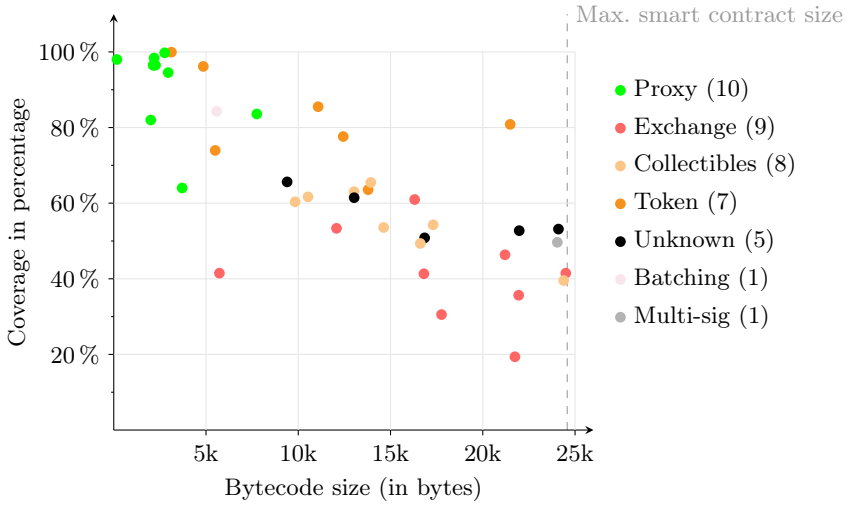

Fig. 6: Scatter plot of bytecode size and code coverage. Timeout 600 seconds.

To complete the validation, we compare `Ethpector` to a concurrent effort. Ma et al. [12] propose a static analysis pipeline starting from source code to find backdoors in token systems. (Recall that `Ethpector` is much more general.) They discover 190 backdoors in deployed ERC20 contracts.[8] We feed the 157 unique addresses on their list into our method. `Ethpector` correctly identifies and extract (at least one) privileged party in 156 cases, suggesting a true positive rate of 99 %. The only contract where `Ethpector` failed[9] to extract the privileged party stores the administrator account in a mapping structure. While we cannot calculate false positives from the biased data, we note that the presence of a privileged party is generally suspicious in an ERC20 contract. For completeness, in the appendix we show `Ethpector`'s console output for the SoarCoin smart contract, the motivating example in [12]. The backdoor therein has caused a loss of $6.6 million to an Australian firm in 2018. `Ethpector` exposes its privileged functions `zero_fee_transaction` and `drain`.

5 Discussion

Next we discuss applications before we move on to limitations in Subsection 5.2.

5.1 Applications

Verifying Zero-Trust. Many smart contract projects start off with a central controlling party and the promise to switch to "zero-trust" in the future. The

[8] `https://github.com/EthereumContractBackdoor/PiedPiperBackdoor/blob/main/Backdoor_List.md`, Accessed 18 Oct 2022.

[9] Ethereum address `0xa821f14fb6394e82839f5161f214cacc90372453`.

transition from central control to "zero-trust" is typically conducted by setting the owner to an address known not to be controlled by any party e. g., zero. `Ethpector` can identify where in the storage of the smart contract the owners is stored. With this information, changes in ownership can be tracked with the information from an Ethereum node. Our method does not require any knowledge about the smart contract, such as known interfaces or self-reporting.

Figure 7 demonstrates this by plotting all changes of privileged parties for all contracts in our validation dataset. Only one smart contract of our validation dataset (the SAITAMA Token) changed its owner to "zero-trust" on 7 July 2021, 7 days after creation.[10]

Audits. Auditors are often tasked to verify the opposite of "zero-trust." Companies that offer services in the Ethereum ecosystem might want to declare their smart contract operations as intangible assets on their conventional balance sheet. To attest that an entity actually "owns" a deployed smart contract, auditors could verify if this is reflected in privileged access to managing functions of the smart contract. Our method allows third parties to do exactly this in an automated manner, i. e., without costly review of the source code.

Detecting Privileged Parameters. For all privileged functions detected with our method, `Ethpector` can extract the storage slots to which the function writes. Slots that are written by one privileged function and read by other (not privileged) functions indicate the existence of a privileged parameter. The analysis results for the SoarCoin contract in Fig. 8 (appendix) demonstrates this. The function `set_centralAccount` is privileged and the only one to write into storage slot 3, suggesting that it holds a privileged parameter. Access to the critical function `zero_fee_transaction`, which reportedly enabled the scam, is controlled by this slot.

Detecting Update Privileges. `Ethpector` also extracts calls and their call targets from the binaries. This information in combination with the check for parameter changes can be used to detect updatable proxies, the common way to keep smart contract code maintainable (and break the immutability feature). More specifically, a smart contract with a privileged function to change an address field that is in turn used to redirect calls is likely used to make the contract updatable. Watching this address field over time allows us to track code updates. Note that our method does not rely on knowledge about the exact update pattern used.

Address Clustering and Network Analysis. For many types of cryptoasset analytics, including forensic investigations, it is useful to lift the unit of analysis from the individual address level to the level of real-world entities. Address clustering is concerned with inferring which addresses belong to the same entity.

[10] The code for generating the figure can be found at `https://github.com/uibk-ethpector/ethpector/blob/main/experiments/privileged-parties/paper/storage_evolution.py`.

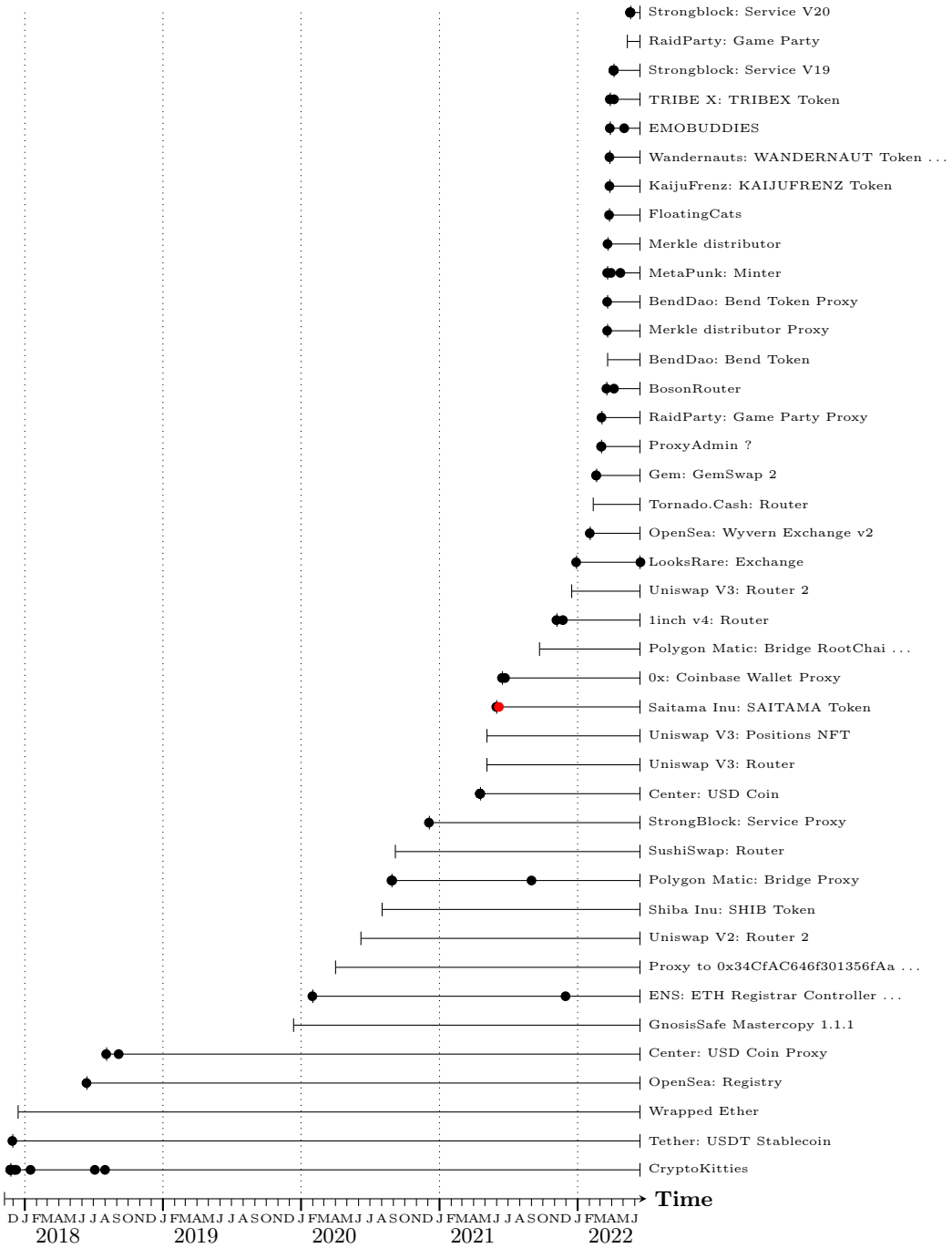

Fig. 7: Changes of privileged parties over time in the validation dataset. Bars are contract lifetimes; black dots indicate changes of privileged parties; red dots (only one in the chart) indicate change to zero, a convention for "zero-trust."

Detecting privileged parties enables to automatically consider smart contracts as well as their controlling accounts as one entity. Similarly, if the same party is detected to have privileges in multiple smart contracts, they can be treated as one conglomerate. `Ethpector` enables a range of applications to study dependencies between smart contracts on the level of their governance structures, potentially uncovering networks of virtual shell organizations.

5.2 Limitations

Obviously our approach has some limitations. We start with conceptual limits before we discuss the most salient technicalities.

Conceptual. A common pattern to distribute the control over global parameters of the system are governance tokens (e. g., Uniswap, MakerDAO). Voting protocols enable the token-holders to jointly make system-wide decisions. While a concentration of tokens in a single party could effectively fulfill our definition of a privileged party, we do not consider this problem here.

We also caution against premature conclusions when `Ethpector` detects a privileged party. The existence of a privileged party does not always imply central control. Exceptions may occur when the identified party is the address of a multi-signature wallet. If such a wallet is realized off-chain, we cannot distinguish it from any other externally owned account using blockchain data only.

Technical. Symbolic execution faces well-known limitations in practice: path explosion, unbounded loops, and the NP-hardness of the SMT problem all require tradeoffs, such as imposing timeouts and skipping paths [2].

The method as proposed fails to detect certain complex administration patterns involving more than one owner (e. g., multi-sig). The challenging smart contracts use advanced data structures in order to manage parties and assign roles to them. While some of the structures could in principle be unrolled, `Ethpector` skips them in the interest of avoiding false positives. This is because the data structures resemble those commonly used to manage token balances. Traversing them would incur the risk that many token functions could be falsely identified as granting privileged access. After all, token holders *are* privileged parties compared to non-holders; but this is not what `Ethpector` is designed to look for. A particular challenging data structure are mappings (i. e., hash tables). Often the address is used as key; and we cannot extract any value before knowing the address we are looking for. As Ethereum nodes do not support enumerating all keys, one would have to customize a node to keep track of all keys so that the correct key–value pair can be known in the analysis. `Ethpector` handles these cases by detecting the existence of a privileged party but fails to identify it.

Finally, our method inspects one smart contract at a time. It cannot resolve authorization patterns that request the privileged party (or other access control information) from other smart contracts using function calls. If this type of authorization becomes more common, the method can be adapted to be aware of common interfaces. This is easier to realize than executing inter-contract communication symbolically.

6 Related Work

This work connects to a number of research strands. We focus on the most relevant ones.

Governance of Cryptocurrency Platforms. Azouvi et al. [1] explore the governance structure of Bitcoin and Ethereum by reviewing the community of contributors to the respective public source-code repositories and discussion boards. They find that only a small number of contributors make up for most of the discussions and code contributions. This suggests a rather centralized governance structure at the level of the development workflow. The authors acknowledged that the centralization seen in the source-code contributions does not directly imply strong centralization of decision making in general. In particular cryptocurrencies diversify control by having miners or validators as gatekeepers.

In contrast to [1], our work is not concerned with the governance structures of the cryptocurrency platforms themselves, but the governance of user-defined applications running on these platforms. Unlike platforms, smart contracts do not have a built-in gatekeeper. The developers alone decide how to distribute control. For example, a significant share of updatable smart contracts are controlled by a single party [13]. This suggests that the governance structures on the Ethereum application layer could be a lot more centralized than the one of the underlying cryptocurrency platform.

Immutability and Code Updates. Fröwis and Böhme [9] study the immutability of the control flow of smart contracts using static program analysis. Using heuristics, they estimate that, in 2017, 40 % of the code accounts on Ethereum host program code that could change its some parts of its code by updating dynamic references. Medhi et al. [13] review approaches to update smart contract implementations. They distinguish between retail and wholesale changes. The former affect parameters; the latter replace the complete implementation. Their measurements focuses on wholesale changes, in particular updates made possible through dynamic references between smart contracts. Between September 2020 and July 2021 they find that in almost 50 % of the identified contracts, a single externally owned account is authorized to update. More recently, Fröwis and Böhme [10] explore a novel way to update code that became available on Ethereum after the Constantinople platform upgrade in early 2019. Using a heuristic indicator, they find more than 100k "potentially updatable" accounts. However, only 41 accounts actually got updated using this method.

Our work is concerned with the more general task of identifying functions of a smart contract that can only be executed by a privileged party. The most popular form of code updates via dynamic references relies on privileged functions only an authorized administrator account can call. Therefore, our method can be applied to detect updatable code. While `Ethpector` cannot extract the indicators used in [10], it is arguably more reliable in detecting the preconditions for such updates than the heuristic approach. The most relevant precondition is whether a `SELFDESTRUCT` instruction is reachable in the execution path.

Backdoors in Token Systems. In concurrent work, Ma et al. [12] propose a static analysis pipeline starting from source code and tailored to identify backdoors in smart contracts implementing token systems. The authors develop detectors for five common backdoors observed in the wild and evaluate their accuracy using active fuzzing. All of the backdoors studied by them require the existence of a privileged party. Although `Ethpector` cannot identify the exact conditions that invoke a backdoor, it is capable of identifying privileged functions. Our method requires bytecode only and is not limited to a specific type of smart contract.

7 Conclusion

We have proposed, implemented, and validated a method to automatically extract privileged parties from EVM binaries using symbolic execution. Our artifact, `Ethpector`, is available on GitHub[11] under an open-source license. It enables a range of applications in research and practice, including the verification of claims about "zero-trust," revelation of ownership structures, and support for forensic analyses of networks of virtual shell organizations. While it cannot replace a thorough code review, it can substantially speed up independent sanity checks; or make them available to audiences who cannot perform a code review.

Future work could refine the detection of more complex authorization patterns, notably boolean expressions involving multiple conditions and privileged parties stored in mapping structures. Both should help to complete the picture by eliminating the (still quite small) gray areas in Figure 3, where `Ethpector` is not yet applicable.

Acknowledgements

This work has received funding from the Austrian Research Promotion Agency (FFG) and the Austrian Security Research Programme (KIRAS).

References

1. Azouvi, S., Maller, M., Meiklejohn, S.: Egalitarian Society or Benevolent Dictatorship: The State of Cryptocurrency Governance. In: Zohar, A., Eyal, I., Teague, V., Clark, J., Bracciali, A., Pintore, F., Sala, M. (eds.) Financial Cryptography and Data Security, BITCOIN Workshop. pp. 127–143. Springer Berlin Heidelberg, Berlin, Heidelberg (2019)
2. Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I.: A Survey of Symbolic Execution Techniques. ACM Comput. Surv. 51(3) (may 2018)
3. Bratspies, R.M.: Cryptocurrency and the Myth of the Trustless Transaction. Michigan Telecommunications and Technology Law Review 25, 1 (2018)
4. Chen, T., Zhang, Y., Li, Z., Luo, X., Wang, T., Cao, R., Xiao, X., Zhang, X.: Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum. In: Proceedings of the 2019 ACM SIGSAC conference on computer and communications security. pp. 1503–1520 (2019)

[11] `https://github.com/uibk-ethpector/ethpector`

5. Chen, W., Zhang, T., Chen, Z., Zheng, Z., Lu, Y.: Traveling the Token World: A Graph Analysis of Ethereum ERC20 Token Ecosystem, p. 1411–1421. Association for Computing Machinery, New York, NY, USA (2020)
6. Di Angelo, M., Salzer, G.: Tokens, Types, and Standards: Identification and Utilization in Ethereum. In: 2020 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS). pp. 1–10. IEEE (2020)
7. Di Angelo, M., Salzer, G.: Identification of Token Contracts on Ethereum: Standard Compliance and Beyond. International Journal of Data Science and Analytics pp. 1–20 (2021)
8. Di Angelo, M., Salzer, G.: Towards the Identification of Security Tokens on Ethereum. In: 2021 11th IFIP International Conference on New Technologies, Mobility and Security (NTMS). pp. 1–5. IEEE (2021)
9. Fröwis, M., Böhme, R.: In Code We Trust? Measuring the Control Flow Immutability of All Smart Contracts Deployed on Ethereum. In: Garcia-Alfaro, J., Navarro-Arribas, G., Hartenstein, H., Herrera-Joancomartí, J. (eds.) Data Privacy Management, Cryptocurrencies and Blockchain Technology, ESORICS 2017 International Workshops. Lecture Notes in Computer Science, vol. 10436, pp. 357–372. Springer, Cham (2017)
10. Fröwis, M., Böhme, R.: Not All Code are Create2 Equal. In: 6th Workshop on Trusted Smart Contracts, associated with Financial Cryptography and Data Security (forthcoming). Lecture Notes in Computer Science, Springer (2022)
11. King, J.C.: Symbolic Execution and Program Testing. Communications of the ACM 19(7), 385–394 (1976)
12. Ma, F., Ren, M., Ouyang, L., Chen, Y., Zhu, J., Chen, T., Zheng, Y., Dai, X., Jiang, Y., Sun, J.: Pied-Piper: Revealing the backdoor threats in Ethereum ERC token contracts. Transactions on Software Engineering and Methodology (2022)
13. Mehdi Salehi, J.C., Mannan, M.: Not so Immutable: Upgradeability of Smart Contracts on Ethereum. In: Financial Cryptography and Data Security, WTSC Workshop. Lecture Notes in Computer Science, Springer (2022)
14. Somin, S., Gordon, G., Altshuler, Y.: Network Analysis of ERC20 Tokens Trading on Ethereum Blockchain. In: Morales, A.J., Gershenson, C., Braha, D., Minai, A.A., Bar-Yam, Y. (eds.) Unifying Themes in Complex Systems IX. pp. 439–450. Springer, Cham (2018)
15. Victor, F., Lüders, B.K.: Measuring Ethereum-based ERC20 Token Networks. In: Goldberg, I., Moore, T. (eds.) Financial Cryptography and Data Security. Lecture Notes in Computer Science, vol. 11598. Springer, Berlin Heidelberg (2019)
16. Vidan, G., Lehdonvirta, V.: Mine the Gap: Bitcoin and the Maintenance of Trustlessness. New Media & Society 21(1), 42–59 (2019)
17. Zhang, L.: Your CryptoKitty Isn't Forever — Why DApps Aren't as Decentralized as You Think (2017), `https://medium.com/loom-network/your-cryptokitty-isnt-forever-why-dapps-aren-t-as-decentralized-as-you-think-871d6acfea`, Accessed on 31 December 2021

A Validation Data

Table 2: Composition of our validation dataset.

	Name	Address	Proxy Implementation	Code Coverage	Type
0	0x: Coinbase Wallet Pro...	0xe66b31678d6c16e9ebf358268a790b763c133750	-	0.82	proxy
1	1inch v4: Router	0x1111111254fb6c44bac0bed2854e76f90643097d	-	0.46	exchange
2	BendDao: Bend Token	0x02863c14603c3b157379999f567ddece151e9153	-	0.73	token
3	BendDao: Bend Token Pro...	0x0d02755a5700414b26ff040e1de35d337df56218	0x02863c ... 151e9153	0.96	proxy
4	BosonRouter	0x0a393aef6dbcd7e7088acf323f9d28b093b9ab5a	-	0.60	exchange
5	Center: USD Coin	0xa2327a938febf5fec13bacfb16ae10ecbc4cbdcf	-	0.81	token
6	Center: USD Coin Proxy	0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48	0xa2327a ... bc4cbdcf	0.98	proxy
7	CryptoKitties	0x06012c8cf97bead5deae237070f9587f8e7a266d	-	0.75	token
8	EMOBUDDIES	0xc995756e8e6a319f209623d0c8f7629dc5636129	-	0.62	collectibles
9	ENS: ETH Registrar Cont...	0x283af0b28c62c092c9727f1ee09c02ca627eb7f5	-	0.66	ens
10	FloatingCats	0xa514ede519870c62d35130ca18e4134e86244255	-	0.60	collectibles
11	Gem: GemSwap 2	0x83c8f28c26bf6aaca652df1dbbe0e1b56f8baba2	-	0.54	collectibles
12	GnosisSafe Mastercopy 1...	0x34cfac646f301356faa8b21e94227e3583fe3f5f	-	0.49	multisig
13	KaijuFrenz: KAIJUFRENZ ...	0xc92090f070bf50eec26d849c88a68112f4f3d98e	-	0.52	collectibles
14	LooksRare: Exchange	0x59728544b08ab483533076417fbbb2fd0b17ce3a	-	0.41	exchange
15	Merkle distributor	0x7529834a5974e2d5fff3d0f0591e9a5ca2ca1619	-	0.84	batching
16	Merkle distributor Prox...	0x1b5d2904be3e4711a848be09b17dee89e6a5bc27	0x752983 ... a2ca1619	0.96	proxy
17	MetaPunk: Minter	0x67401149e3e88b10dd92821eb6302f4dee8191bc	-	0.60	other services
18	OpenSea: Registry	0xa5409ec958c83c3f309868babaca7c86dcb077c1	-	0.64	proxy
19	OpenSea: Wyvern Exchang...	0x7f268357a8c2552623316e2562d90e642bb538e5	-	0.19	exchange
20	Polygon Matic: Bridge P...	0xa0c68c638235ee32657e8f720a23cec1bfc77c77	0x6abb75 ... 105fd5f5	0.95	proxy
21	Polygon Matic: Bridge R...	0x6abb753c1893194de4a83c6e8b4eadfc105fd5f5	-	0.51	other services
22	Proxy to 34cfac64 ... 8...	0xe24f4870ab85de8e356c5fc56138587206c70d99	0x34cfac ... 83fe3f5f	0.98	proxy
23	ProxyAdmin ?	0x1f0e8a86e398f0f2fd285a36d2f1ee85d6bbc9c5	-	1.00	proxy
24	RaidParty: Game Party	0x62e4760e59ce31865a09cfc7cdc27432eb4433db	-	0.49	collectibles
25	RaidParty: Game Party P...	0xd311bdacb151b72bddfee9cbdc414af22a5e38dc	0x83d0c1 ... 1ac57fcc	0.82	proxy
26	Saitama Inu: SAITAMA To...	0x8b3192f5eebd8579568a2ed41e6feb402f93f73f	-	0.64	token
27	Shiba Inu: SHIB Token	0x95ad61b0a150d79219dcf64e1e6cc01f0b64c4ce	-	0.96	token
28	StrongBlock: Service Pr...	0xfbddadd80fe7bda00b901fbaf73803f2238ae655	0xdcbf1e ... 9d5a41cf	0.96	proxy
29	Strongblock: Service V1...	0xc2899dfcb0a81b73e89e4a99cd24ab26d8a78295	-	0.53	other services
30	Strongblock: Service V2...	0x88f6ed8fb519612a03730e4b5a5c1581a7d0c305	-	0.54	other services
31	SushiSwap: Router	0xd9e1ce17f2641f24ae83637ab66a2cca9c378b9f	-	0.34	exchange
32	TRIBE X: TRIBEX Token	0xff9981d2c6c6d612e03e4a32f5488e552eeae285	-	0.63	collectibles
33	Tether: USDT Stablecoin	0xdac17f958d2ee523a2206206994597c13d831ec7	-	0.86	token
34	Tornado.Cash: Router	0xd90e2f925da726b50c4ed8d0fb90ad053324f31b	-	0.41	privacy
35	Uniswap V2: Router 2	0x7a250d5630b4cf539739df2c5dacb4c659f2488d	-	0.36	exchange
36	Uniswap V3: Positions N...	0xc36442b4a4522e871399cd717abdd847ab11fe88	-	0.40	collectibles
37	Uniswap V3: Router	0xe592427a0aece92de3edee1f18e0157c05861564	-	0.54	exchange
38	Uniswap V3: Router 2	0x68b3465833fb72a70ecdf485e0e4c7bd8665fc45	-	0.41	exchange
39	Wandernauts: WANDERNAUT...	0x793daf78b74aadf1eda5cc07a558fed932360a60	-	0.67	collectibles
40	Wrapped Ether	0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2	-	1.00	token
	TOTAL:			0.66	

B `Ethpector` in a Nutshell

The source-code of `Ethpector` is publicly available and can be acquired from GitHub.[12] `Ethpector`'s main focus is the analysis and exploration of EVM compatible binaries. Given some binary or the on-chain address the tool provides an annotated view on the disassembled program. Currently, the tool supports two static analysis techniques to extract data from the binaries. First, it implements abstract interpretation to produce the reaching definitions for each position in the program. The reachings are then used to construct the control flow graph of the program. Additionally, the tool includes Mythril,[13] a security analysis tool for EVM-binaries. `Ethpector` uses Mythril's symbolic execution engine to

[12] `https://github.com/uibk-ethpector/ethpector`

[13] `https://github.com/ConsenSys/mythril`, Accessed: 07 June 2022

annotate the binaries with e. g., privileged functions, logs and functions as well as their parameters.

In addition to the capabilities to automatically extract data from EVM binaries the tools integrates with several online sources to improve the analysis workflow. It provides Etherscan integration to fetch smart contract code and meta-data. Additionally, source-code can be fetched from Sourcify,[14] a decentralized platform to publish smart contract source-code. This enables quick an easy access to source code, if available. To recover the public interface of a smart contract even if source-code is not available `Ethpector` uses the local signature lookup table provided by Mythril as well as the online services 4bytes and etherface.io.[15] This enables the reconstruction of public interfaces of the binaries as long as the function and event signatures are publicly known.

Notable features:

- Reconstruction of jump targets using data flow analysis and symbolic execution.
- Detection of known and standardize function interfaces.
- Html, Ti*k*Z and NetworkX privileged parties graph output.
- Identification of standard proxies and extraction of implementation contracts
- Extraction of calls and their parameters.
- Extraction of logs and their parameters.
- Extraction of storage reads and writes including their positions.
- Fully featured parser for function and event definitions and parsing of corresponding logs and inputs.
- Reconstruction of Solidity revert messages.

The code for the experiments conducted in this paper as well as the exact configuration used, can be found in the experiment folder in the repository. Figure 8 shows the output of the overview command line UI of `Ethpector`.[16]

C The Ownership Pattern

The most common pattern to encode privileged parties is called the ownership pattern. Although, there exists no official standard to encode ownership, canonical implementations are readily available in popular code libraries.[17]

The ownership pattern implements a simple function level authorization structure. The owner can be changed via the `changeOwner` function. The owner

[14] `https://sourcify.dev/`, Accessed: 07 June 2022

[15] `https://www.4byte.directory/` and `https://www.etherface.io`, Accessed: 07 June 2022

[16] For address `0xD65960FAcb8E4a2dFcb2C2212cb2e44a02e2a57E`. The code and entry point for the overview UI can be found in the folder `experiments/risk_report` in the repository.

[17] `https://docs.openzeppelin.com/contracts/2.x/api/ownership`, Accessed 07 June 2022.

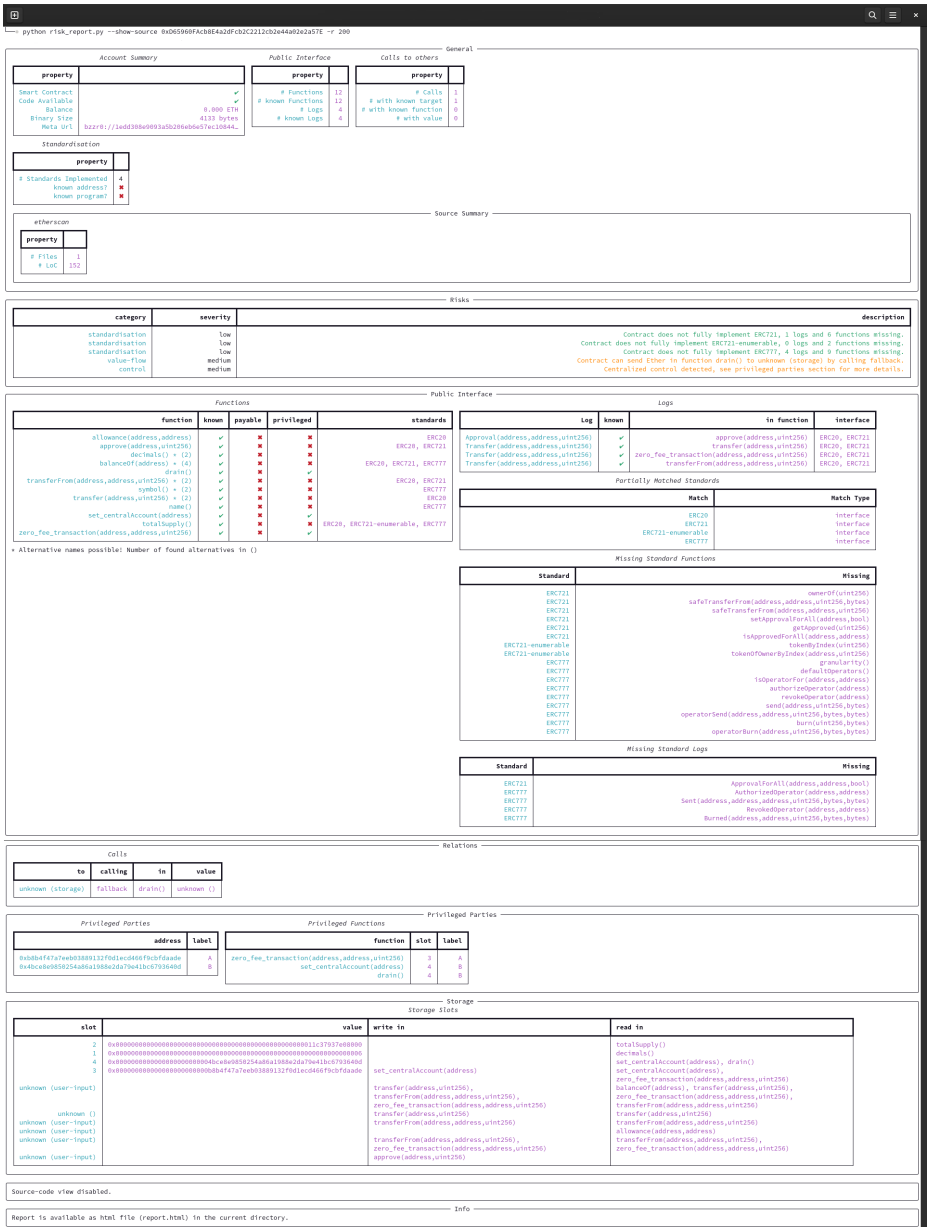

Fig. 8: Example output of the `Ethpector` console UI. The address under analysis is the SoarCoin smart contract. Its privileged function `zero_fee_transaction` is a backdoor that caused a loss of $6.6 million to an Australian firm [12].

is stored in an dedicated slot in permanent storage.[18] Functions that should be exclusive to the owner are annotated with the modifier `isOwner`. This instructs the compiler to insert the ownership check in the preamble of the function. If the function is executed by a sender other than the current owner the transaction reverts.

```
contract Owner {

    address private owner;

    event OwnerSet(address indexed oldOwner, address indexed newOwner);

    // modifier to check if caller is owner
    modifier isOwner() {
        require(msg.sender == owner, "Caller is not owner");
        _;
    }

    constructor() {
        owner = msg.sender;
        emit OwnerSet(address(0), owner);
    }

    function changeOwner(address newOwner) public isOwner {
        emit OwnerSet(owner, newOwner);
        owner = newOwner;
    }

    function getOwner() external view returns (address) {
        return owner;
    }
}
```

Listing 1.1: Ownership pattern. The common form of defining privileged parties.

[18] Note: The address of the storage slot is dependent on the storage fields defined earlier in the contract.