ContractViz: Extending Eclipse Trace Compass for Smart Contract Transaction Analysis

Xiaolin Liu
KTH Royal Institute of Technology
Stockholm, Sweden
ORCID: 0009-0008-2922-7870

Adel Belkhiri
École Polytechnique de Montréal
Montreal, Canada
ORCID: 0000-0001-6299-5574

Mónica Jin
KTH Royal Institute of Technology
Stockholm, Sweden
ORCID: 0009-0000-8775-2093

Yi Li
Nanyang Technological University
Singapore, Singapore
ORCID: 0000-0003-4562-8208

Cyrille Artho
KTH Royal Institute of Technology
Stockholm, Sweden
ORCID: 0000-0002-3656-1614

***Abstract*—The complexity of the Ethereum smart contracts makes it challenging to avoid security flaws. This problem led to many code analysis tools, which detect potential flaws and report them textually. However, the lack of context and visual information in these reports hinders the stakeholders' understanding of the detailed information. Visualization can assist a developer in grasping such context, but current state-of-the-art visualization tools provide only fixed and limited visualization types. To this end, we present ContractViz, based on the versatile platform Eclipse Trace Compass (TC), which supports various views and analyses in parallel. Our contribution enables TC to visualize Ethereum transaction traces using flame charts and gas consumption plots. This reveals information on account activities and provides insights into the correct or possibly flawed behaviors.**

***GitHub repo*—https://github.com/AisXiaolin/ContractViz**

***YouTube video*—https://aisxiaolin.github.io/VideoDemo/**

***Index Terms*—Blockchain, Smart Contracts, Visualization, Eclipse Trace Compass, Transaction Logs**

I. Introduction

Smart contracts manage information and assets using blockchain technology, providing immutable pieces of software with unique self-executable properties [1]. However, security issues have become concerning since they frequently lead to financial loss. One infamous security incident is the DAO attack, which ultimately led to a loss of around $ 60 million due to the vulnerability caused by a reentrancy attack on an Ethereum smart contract [2].

The root problem is that smart contracts often have exploited flaws, resulting in financial loss, including reentrancy attacks [3], infinite loops [4] and gas inefficiency problems [5]. These flaws exist because many smart contract functionalities are intricate and complex (e. g., callbacks [6], external contract calls [3], or dependencies on block timestamps [7]).

To detect such flaws early in development, ideally, before a contract is deployed as immutable software on a production blockchain, many vulnerability analysis tools have been developed [8]. These analysis tools often find subtle flaws in software, but static (compile-time) analysis tools may over-report problems due to over-approximating all possible outcomes; dynamic analysis tools may miss issues due to no test revealing them. We, therefore, need a way to understand and validate the output of these tools, but that output is usually textual and hard to evaluate, especially against potential false positives [1]. Our method potentially speeds up this process and can also improve trust in diagnostics.

Visualizing execution behavior can help a developer understand the code's functionality or find the root cause of a suspected problem. Our work, ContractViz, shown in Fig.2, complements existing approaches by showing the transaction log details on Eclipse Trace Compass (TC). TC is a mature platform that combines visual analytics with filters and statistics and offers several types of visualization alongside a tabular log representation. ContractViz augments TC with smart-contract visualization and shows the corresponding gas usage, balance changes, or call stack behavior when the transaction is performed, allowing users to interact with detailed execution flows between Uniswap smart contracts and taken contracts like Wrapped Ether and USDC. The added features shown in the foundational capabilities of Eclipse Trace Compass are all uniquely tailored for Smart Contract data analysis.

However, a number of mismatches between traditional programs and smart contracts prevent a straightforward visualization of TC. First, traditional programs (and TC) do not have a notation of account balances or gas fees, which are prevalent in smart contracts. Second, TC expects all events to be annotated by a time stamp taken from a system clock, but this information is not available for blockchain transactions. Third, TC expects the data in specific formats.

In the following sections, we will discuss our method and tool in more detail. We extract the necessary data and fill it into the required data fields in TC's trace event format. ContractViz can show (1) gas consumption over time and (2) detailed function call traces. The above insights also provide insight into solving inefficiencies, out-of-gas issues, reentrancy vulnerabilities, and infinite loops.

Fig. 1. Data Retrieval, Processing and Visualization

II. Background

A. Smart contracts

Smart contracts are self-executable programs operating on blockchain platforms when predefined conditions are met. Once deployed on a blockchain platform like Ethereum, a contract is immutable and can be invoked by sending transactions to its address. Its state can be modified through transactions and is recorded in the blockchain's ledger [1].

The decentralized nature of smart contracts ensures that no single entity controls the contract's execution and governs access [6]. Gas fees were introduced to ensure that computational resources are allocated efficiently and to prevent the abuse of resources required to manage smart contracts. Unlike Bitcoin, Ethereum is Turing-complete, and gas fees are used to solve the challenges related to the halting problems. An Ethereum smart contract will halt its execution once the gas fee has been used, preventing the program from entering infinite loops and consuming excessive amounts of resources [9]. Callbacks in smart contracts occur when the contract calls a function in another contract. The most concerning vulnerability in this context is the reentrancy attack, where an unintended loop changes the state of a contract by allowing the external contract to alter the internal state unexpectedly [3], [6].

B. Eclipe Trace Compass

Eclipse Trace Compass[1] (TC) is open-source software for analyzing the traces and logs of a system. It provides different views, graphs, and metrics to help extract informative information from traces. Developers can interact with the visualized data across different forms of representations. TC also supports statistical analysis over trace data.

III. Data Preparation and Software Setup

A. Ethereum Smart Contract Trace Extraction

Ethereum Virtual Machine (EVM) traces are not stored directly on the blockchain; rather, they are generated on demand by an Ethereum node. To collect real smart contract traces, we deploy an Erigon[2] node. Erigon is an optimized Ethereum client that supports high-performance data retrieval, making it suitable for our needs. We access the relevant data through a sequence of Remote Procedure Calls (RPC) interactions, which allow us to communicate with the node programmatically.

[1]https://eclipse.dev/tracecompass/
[2]https://erigon.tech/

TABLE I
Data Mapping Table

Source Field	Mapped Field
traces.callType	traceEvent.name
blockNumber	traceEvent.pid
traces.gas	args.gasUsed

We use the `trace_transaction` RPC call to collect traces for a given transaction. This call replays a transaction to provide a detailed log of each internal operation, including calls and value transfers. For temporal context, we retrieve the block timestamp associated with the transaction using the `eth_getBlockByNumber` RPC call, specifying the block number.

To illustrate this process, consider the transaction shown in Fig.2. The hash number was carefully selected for our analysis to have more detailed internal transactions. The transaction hash[3] shown represents a transaction from block number `14157500`.

B. Mismatches between smart contract data and regular execution traces

Eclipsed Trace Compass requires the input JSON file to have a trace event format so that the software can parse the input data into log traces. TC expects seven mandatory data items, which include `name`, `cat`, `ph`, `pid`, `tid`, `ts`, `args`, representing a trace event's name, category, phase, process ID, thread ID, timestamp and other arguments, respectively. However, events in smart contracts are not timestamped; instead, they are added to a block with a specific number, and that block number increases monotonically over time. Smart-contract data also includes items like gas usage, which TC does not support.

C. Conversion of EVM traces to TC's Trace Event Format

Our conversion converts each smart contract event to support all mandatory fields while allowing additional data to be attached to events. This conversion involves two steps:

First, we map event fields to TC's expected structure where possible, reusing TC's functionality. Table I shows three entries that can be directly matched from the traces' raw data. Furthermore, we have to convert a single function call to a triple of data items with these expected seven fields: Each function call is represented by its duration and type, with event type `B` and `E` representing the beginning and end of the event and the middle `C` representing the counter events. The timestamps for these three events have been designed so that the function call at a higher hierarchy has a longer duration to cover its sub-traces.

We use attribute `call` for all `traceEvent.cat` items, since we do not need `subcall` to show any branch-out functions. The number for `traceEvent.pid` is

[3]0x55a72f6c7608257afed88cd423e050368c0e3b2cba94a23c51ab811827b89f01

the `blockNumber` from the raw data.[4] We manually adjust the `traceEvent.tid` so that the correct hierarchy represents the correct parent and child function call. The gas fee has been directly imported to the `args.gasUsed`, making it one of the optional domain-specific types of parameters that we can supply to TC.

Second, we calculate the timestamps for different event types. We converted the block deployed time to UNIX timestamp format and inserted it in the "End" event type for the last function call. The timestamp is then calculated backward from that point. For simplicity, we assign a unit time duration to leaf nodes.

D. Trace Compass Data-driven Analyses

TC comes equipped with several pre-built analyses, enabling the efficient processing of the collected traces and the extraction of important information, thus providing the user with valuable insights into the traced system. The extracted information is often plotted as time-synchronized views, graphs, and charts. For custom analysis development, TC provides different methods tailored to the user's expertise and different levels of analysis complexity. For instance, users can develop analyses as Java modules or leverage Eclipse Advanced Scripting Environment (EASE) to implement the analysis in JavaScript or Python. Moreover, TC supports the development of analyses in XML code, thus enabling users to define rules for data extraction and processing directly from the traces. The data generated by custom analyses are often visualized using time graphs and XY chart views. An XY chart displays data series as numerical values plotted over time. The X-axis represents time, while the Y-axis can represent any numerical value that is relevant to the analysis (e.g., gas consumption). On the other hand, a time graph view is divided into two sections: a tree viewer on the left displaying different entries and a Gantt-like viewer on the right that shows the state of these entries over time. Fig. 2 shows a flame chart, which is an example of a time-graph view, whereas the chart below representing the variation of gas fee per function type is an example of an XY chart.

We have developed several analyses to show the variation of gas consumption from different perspectives, such as per function type and per transaction. As the scripting and XML-based methods enable users to share and run analyses in TC without the need for recompilation, we chose to implement our analyses using XML code. Our "Ethereum Fee per Function" view, for instance, displays time on the X-axis and gas cost on the Y-axis. Different types of function calls are represented by lines of different colors. Once a function type is selected, the chart updates to reflect changes in gas cost when calls to this function type are triggered and result in different gas consumption. Even when multiple calls for the same function type overlap, the chart displays the specific gas usage for the newly triggered function call.

[4]We use the `callType` as `traceEvent.name` and the `blockNumber` as `traceEvent.pid` so they appear as labels in suitable places.

IV. Visual Ethereum Call Trace Analytics

ContractViz will help users and developers understand smart contracts and their transactions. It generates a synthetic timestamp for each event depending on its block's timestamp and the execution order. This meets TC's requirements for timestamped trace logs for a detailed visualization of smart contract interactions. The analysis results show the potential to improve transaction security concerns; below, we discuss the analysis of gas usage (which can highlight out-of-gas risks [5] in specific portions of the code) and detailed call traces (which can show unexpectedly deep recursion, indicative of a potential reentrancy [3] vulnerability).

A. Gas Consumption Over Time

In a smart contract on Ethereum, the gas cost corresponds to the resources needed for its execution. Using Eclipse Trace Compass to plot the gas usage for each transaction over time can improve the chances of identifying potential inefficiencies and gas-related vulnerabilities. Once the correct pattern for the particular smart contact type has been confirmed, the gas inefficiencies in the smart contract execution could be directly spotted [5]. Patterns would also be detected when the contract runs out of gas, leading to the transaction's failure or even denial of service [3], [10]. Since Ethereum has also been introduced as a platform that facilitates a Turing-complete programming language to be executed on the network, an Ethereum smart contract must be triggered with payment of the required gas fee to avoid exceeding available computation resources.

Visualizing gas usage helps distinguish and recognize how infinite loops or complex function calls will drain the gas unexpectedly. Correlation shows that the opcodes and source code parameters can directly increase or decrease the gas cost [11]. Developers can deliberately adjust or optimize gas consumption; thus, the gas alone cannot be the sole metric for evaluating the transactions, plus it lacks the sequential context. From our generated XY chart analysis, function calls have been categorized and grouped, comparing the gas cost between different function types and horizontally showing the difference between the same function calls. This supports developers in assessing whether the gas fee has been designed reasonably for efficiency and resource distribution [12].

B. Detailed Function Call Traces

Showing the internal function calls can help reveal reentrancy vulnerabilities, infinite loops, or similar security issues. ContractViz shows transaction log entries as a time series or as various diagrams. The visualized traces show the function call hierarchy inside a transaction, especially which parent function call triggers the child function. The tracked internal function calls can suggest reentrancy vulnerabilities so the developers can detect and spot the functions that have been called multiple times before the previous call is completed. For example, the visualized function call hierarchy on the upper part of Fig. 2 can show infinite loops and recursive calls that might be caused by exhausting the gas of a contract

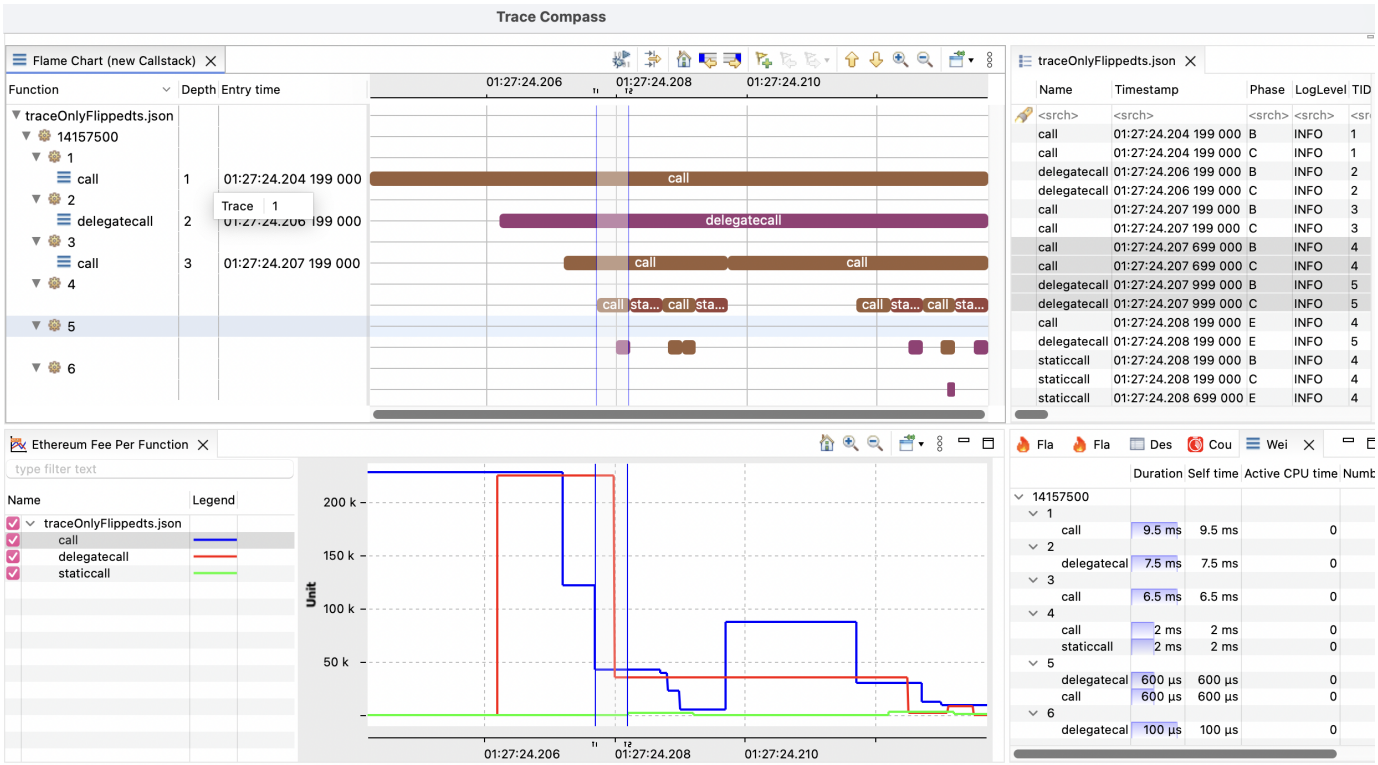

Fig. 2. ContractViz User Interface

or unpredictable contract transaction behavior. Potential access control issues could also be shown in the visualization of the internal function calls. Thus, developers can ensure that the function calls are protected and only authorized parties can access specific system resources [13].

V. Related Work

Different approaches can be used to visualize smart contracts and reveal their behaviors. We give a brief overview of key related works, classifying them as *dashboards* that summarize key data, *model-based* approaches that use user-defined graphical models to represent key functionality, and *graph analysis* approaches where extracted data is represented as a graph without an underlying human-defined model.

Dashboards: A dashboard view is often used as an entry point to smart contract analysis tools to provide an overview of the results [14], [15]. A dashboard can show transaction logs along with application-specific data, such as which data items were used or modified most often. In this way, the application integrates a dashboard as an overview of its internal state and hides low-level details of the smart contract [14]. A dashboard can give access to manage larger code bases [15] to help compare smart contracts and analyze their code. BlockSec[5], for instance, provides the visualization of call stacks similar to the flame chart in TC. However, TC complements this feature, focuses on the run-time behavior of code, and is application-agnostic (while still allowing a user to filter the data to highlight specific items). Unlike BlockSec, TC allows multiple analyses to be shown on one single screen. Instead of embedding hyperlinks to more detailed information that redirect users to another page on BlockSec, TC enables this feature by hiding more detailed data behind the button, and detailed information unfolds only when the mouse hovers above.

Model-based approaches: Model-based approaches use a high-level model to reason about low-level functionality. These models often have a graphical notation, with UML being the most well-known example [16]. To model the structure of smart contract code, SCMTool provides a graphical modeling platform using UML-like class diagrams [17]. The focus lies on how different components relate to and interact with each other. However, structural models often do not represent semantics sufficiently well and may elide important details [16]. To this end, behavioral models have been proposed to model the *behaviour* of smart contracts [18]. Using dynamic condition-response (DCR) graphs, it is possible to capture temporal and logical dependencies of different activities in smart contracts, which can help to understand them better and can also be used to monitor their behavior at run time [19]. Compared to model-based approaches, our work visualizes the direct behavior of the transactions rather than an abstraction or model of them.

Graph Analysis approaches: A graph-based analysis recovers the structure (often also quantitative information such as coverage percentages) from smart contracts without needing

[5]https://blocksec.com/

a human-defined model as a starting point. A systematic study has been conducted on Ethereum by constructing the money flow graph (MFG), smart contract creation graph (CCG), and smart contract invocation graph (CIG) to characterize their activities [20]. The examination result of the individual graph reveals securities issues, including account forensics, anomaly detection, and deanonymization. Similar to the introduced graph metrics that provide users with a better understanding of the properties of Ethereum, TC uses different visualization windows to analyze the contracts' properties. Further, TC extends the graphical features by allowing the developers to navigate the data and interact with the visualizations.

VI. Conclusions and Future Work

Visualization assists developers in collecting informative and intuitive data information. Our tool, ContractViz, complements the current state-of-the-art visualization tools by increasing the fixed and limited visualization types and providing a more versatile platform. By building on Eclipse Trace Compass (TC), ContractViz supports multiple views and analyses in parallel. Our contribution enables TC to visualize Ethereum transaction traces using flame charts for the internal function calls and gas consumption over time. The revealed information on account activities provided insights into the correct or possibly flawed behaviors.

In future work, we want to further integrate and optimize the automated conversion of Ethereum traces and create custom, more compact, or expressive visualizations of that data. Another goal is to visualize multiple transactions simultaneously so that detailed information on the external function calls can also be shown and analyzed. The result will then be organized for the user case study, and the generated result could be worked on to improve our program's performance and effectiveness based on the evaluation results.

Acknowledgment

We would like to thank the Eclipse Trace Compass Developer team from Montréal, which has provided valuable insights and suggestions on the technical part of the project.

References

[1] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, and H.-N. Lee, "Ethereum smart contract analysis tools: A systematic review," *IEEE Access*, vol. 10, pp. 57 037–57 062, 2022.

[2] M. I. Mehar, C. L. Shier, A. Giambattista, E. Gong, G. Fletcher, R. Sanayhie, H. M. Kim, and M. Laskowski, "Understanding a revolutionary and flawed grand experiment in blockchain: the DAO attack," *Journal of Cases on Information Technology (JCIT)*, vol. 21, no. 1, pp. 19–32, 2019.

[3] N. F. Samreen and M. H. Alalfi, "Reentrancy vulnerability identification in Ethereum smart contracts," in *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 2020, pp. 22–29.

[4] T. Osterland and T. Rose, "Model checking smart contracts for Ethereum," *Pervasive and Mobile Computing*, vol. 63, p. 101129, 2020.

[5] Q.-P. Kong, Z.-Y. Wang, Y. Huang, X.-P. Chen, X.-C. Zhou, Z.-B. Zheng, and G. Huang, "Characterizing and detecting gas-inefficient patterns in smart contracts," *J Comput Sci Technol*, vol. 37, no. 1, pp. 67–82, 2022.

[6] S. Sayeed, H. Marco-Gisbert, and T. Caira, "Smart contract: Attacks and protections," *IEEE Access*, vol. 8, pp. 24 416–24 427, 2020.

[7] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of Ethereum smart contracts," in *Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain*, 2018, pp. 9–16.

[8] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, and H.-N. Lee, "Ethereum smart contract analysis tools: A systematic review," *IEEE Access*, vol. 10, pp. 57 037–57 062, 2022.

[9] G. A. Pierro and H. Rocha, "The influence factors on Ethereum transaction fees," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2019, pp. 24–31.

[10] C. Ferreira Torres, A. K. Iannillo, A. Gervais, and R. State, "The eye of Horus: Spotting and analyzing attacks on Ethereum Smart Contracts," in *International Conference on Financial Cryptography and Data Security*. Springer, 2021, pp. 33–52.

[11] M. M. A. Khan, H. M. A. Sarwar, and M. Awais, "Gas consumption analysis of Ethereum blockchain transactions," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 4, p. e6679, 2022.

[12] A. Chepurnoy, V. Kharin, and D. Meshkov, "Self-reproducing coins as universal turing machine," in *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, J. Garcia-Alfaro, J. Herrera-Joancomartí, G. Livraga, and R. Rios, Eds. Cham: Springer International Publishing, 2018, pp. 57–64.

[13] B. Liu, S. Sun, and P. Szalachowski, "Smacs: Smart contract access control service," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2020, pp. 221–232.

[14] S. K. Yap, Z. Dong, M. Toohey, Y. C. Lee, and A. Y. Zomaya, "Smart contract data monitoring and visualization," in *2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, 2023, pp. 1–8.

[15] J. Park, S. Jeong, and K. Yeom, "Smart contract broker: improving smart contract reusability in a blockchain environment," *Sensors*, vol. 23, no. 13, p. 6149, 2023.

[16] G. A. Pierro, "Smart-graph: Graphical representations for smart contract on the ethereum blockchain," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021, pp. 708–714.

[17] G. C. Velasco, D. A. Vieira, M. A. Vieira, and S. T. Carvalho, "Scmtool: A graphical tool for smart contract modeling," in *Anais do I Colóquio em Blockchain e Web Descentralizada*. SBC, 2023, pp. 31–36.

[18] M. Eshghie, W. Ahrendt, C. Artho, T. T. Hildebrandt, and G. Schneider, "Capturing smart contract design with dcr graphs," in *International Conference on Software Engineering and Formal Methods*. Springer, 2023, pp. 106–125.

[19] M. Eshghie, C. Artho, H. Stammler, W. Ahrendt, T. Hildebrandt, and G. Schneider, "Highguard: Cross-chain business logic monitoring of smart contracts," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 2378–2381.

[20] T. Chen, Z. Li, Y. Zhu, J. Chen, X. Luo, J. C.-S. Lui, X. Lin, and X. Zhang, "Understanding Ethereum via graph analysis," *ACM Transactions on Internet Technology (TOIT)*, vol. 20, no. 2, pp. 1–32, 2020.