Informatics

Advancing Symbolic Execution Tools with a Transient Storage Model for Vulnerability Detection in Smart Contracts

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Jaak Weyrich, BSc.
Matrikelnummer 11814804

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Ao.Univ.Prof.Dr. Gernot Salzer
Mitwirkung: Ass.Prof.in Dr.in Monika di Angelo

Wien, 7. August 2024

Jaak Weyrich　　　Gernot Salzer

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

TU WIEN Informatics

Advancing Symbolic Execution Tools with a Transient Storage Memory Model for Vulnerability Detection in Smart Contracts

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Business Informatics

by

Jaak Weyrich, BSc.
Registration Number 11814804

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof.Dr. Gernot Salzer
Assistance: Ass.Prof.in Dr.in Monika di Angelo

Vienna, August 7, 2024

Jaak Weyrich   Gernot Salzer

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Jaak Weyrich, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Overview of Tools Used" habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, habe ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 7. August 2024

Jaak Weyrich

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Diese Arbeit beschäftigt sich mit Transient Storage, welcher als Zwischenspeicher in Smart Contracts dient und Eigenschaften sowohl des flüchtigen Smart Contract Memory (insbesondere die Eigenschaft, nach jeder Transaktion zurückgesetzt zu werden) wie auch des permanenten Smart Contract Storage (insbesondere die Eigenschaft, innerhalb einer Transaktion über mehrere Smart Contract Frames hinweg bestehen zu bleiben) aufweist. Seine korrekte Modellierung ist entscheidend für die präzise Programmanalyse und Schwachstellenerkennung in Smart Contracts. Insbesondere untersucht diese Arbeit die geeignete Modellierung von Transient Storage für Symbolic Execution Tools, mit einem Fokus auf den Trade-off zwischen Genauigkeit und Recheneffizienz bei der Erkennung von Schwachstellen in Smart Contracts. Sie identifiziert Schlüsselfaktoren und deren Implikationen für die Implementierung des Modells, mit dem Ziel, den besten Ansatz zu finden. Wir führen eine vergleichende Analyse der existierenden State-of-the-Art Symbolic Execution Tools durch, gefolgt von der Entwicklung einer Erweiterung für das Symbolic Execution Tool Mythril, die aus der Modellierung des Transient Storage besteht. Die neu entwickelte Erweiterung wird dann systematisch evaluiert, um ihre Effektivität in der Verbesserung der Erkennung von Schwachstellen in Smart Contracts zu bewerten. Die Evaluation zeigt, dass unser Ansatz, ein Python Dictionary statt eines SMT Arrays für die Modellierung des Transient Storage zu verwenden, eine praktikable Alternative darstellt, die Recheneffizienz ohne Einbußen bei der Genauigkeit bietet.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Abstract

This work focuses on Transient Storage, which serves as an intermediary storage in Smart Contracts and exhibits characteristics of both the volatile Smart Contract Memory (notably the property of being reset after each transaction) and the permanent Smart Contract Storage (notably the property of persisting across multiple Smart Contract frames within a transaction). Its accurate modeling is crucial for precise program analysis and vulnerability detection in Smart Contracts. Specifically, this thesis explores the appropriate modeling of transient storage for symbolic execution tools, focusing on achieving the right balance between accuracy and computational efficiency in detecting vulnerabilities in smart contracts. It identifies key factors and their implications for model implementation, aiming to identify the best approach. We conduct a comparative analysis of existing state-of-the-art symbolic execution tools, followed by the development of an extension to the symbolic execution tool Mythril, that consists of transient storage modeling. The newly developed extension is then systematically evaluated to assess its effectiveness in enhancing the detection of vulnerabilities in smart contracts. The evaluation illustrates that our approach, employing a Python dictionary instead of an SMT array for transient storage modeling, presents a viable alternative offering computational efficiency without compromising accuracy.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Kurzfassung **vii**

Abstract **ix**

Contents **xi**

1 Introduction **1**
1.1 Motivation and Problem Statement 1
1.2 Aim of the Thesis and Expected Results 2
1.3 Methodology 2
1.4 Related Work 3

2 Technical Background **5**
2.1 SMT Solver and Theories 5
2.2 Symbolic Execution 6
2.3 Data storage types in smart contracts 6

3 EIP-1153: Transient Storage **9**
3.1 Motivation of Transient Storage 9
3.2 Technical Details of EIP-1153 10
3.3 Use Cases and Applications 10

4 Comparative Analysis of existing Data Storage Modeling Techniques **13**
4.1 Analysis of the tools 13
4.2 Analysis and Comparison of Methods 15
4.3 Additional observations 17

5 Definition of Requirements **19**
5.1 Modeling of storage as baseline 19
5.2 Requirements towards accuracy 19
5.3 Requirements towards efficiency 20

6 Development: Modeling Transient Storage **21**
6.1 Extension of Mythril 22

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

6.2 Development Environment and Tools 22
6.3 Design Choices and Implementation Details 22
6.4 Mythril Analysis Modules 26

7 Demonstration **27**
7.1 Setup 27
7.2 Executing the Tool 28
7.3 Demonstration Contracts 29

8 Evaluation **35**
8.1 Contracts used for evaluation 35
8.2 Preparation of evaluation contracts 35
8.3 Accuracy 36
8.4 Efficiency 36
8.5 Hardware Setup 38
8.6 Limitations of the Evaluation 38

9 Discussion **41**
9.1 Implications of Findings 41
9.2 Future Research 42

List of Figures **45**

List of Tables **47**

List of Smart Contracts **49**

Code and Data Availability **51**

Overview of Tools Used **53**

Bibliography **55**

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 1

Introduction

1.1 Motivation and Problem Statement

The Ethereum platform [But14], which enables the execution of autonomous programs, called smart contracts, on a decentralized ledger, has witnessed a strong increase in number of applications and use cases. Accordingly, the high financial throughput makes smart contracts an attractive target for malicious actors. Like other kinds of software, smart contracts are prone to bugs, and exploits can lead to high financial damage [Cha23].

Symbolic execution is a technique for strengthening the security of smart contracts by attempting to explore all possible paths through a program, thereby enabling a comprehensive coverage of the program's behavior [Kin76]. For the symbolic execution to provide results efficiently and accurately, the different data storage areas of the Ethereum Virtual Machine (EVM), the smart contracts' runtime environment, need to be modeled appropriately in the symbolic execution tool. The modeling needs to be precise enough to catch important details but abstract enough to provide efficiency. This is no easy task since the EVM data storage architecture is quite complex, incorporating many different types of data storage. Making the wrong decisions can lead to problems such as path explosion or a large number of false positives.

Additionally, the EVM data storage architecture is expected to soon incorporate *transient storage* [AS18], complicating the landscape further. This new data storage type functions similarly to a smart contract's existing *storage* but is ephemeral, with its contents being discarded after the transaction completes. No tools publicly available currently account for 'transient storage'. Also, not all symbolic execution tools for smart contracts are accompanied by a scientific paper, and for those that are, the paper usually does not provide a profound reasoning behind the data storage modeling. This underscores the need for a robust implementation that includes a detailed analysis and justification of

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

how transient storage should be modeled within symbolic execution tools.

1.2 Aim of the Thesis and Expected Results

The aim of this thesis is to provide an extensive examination and reasoning of how transient storage should be modeled in a symbolic execution tool to discover vulnerabilities in smart contracts. The thesis will be accompanied by the implementation of the data storage modeling for transient storage, leveraging the insights gained in the afore-mentioned examination. Specifically, the tool Mythril [Con18] will be extended in this regard.
We aim to answer the following research questions:

- Which modeling techniques do state-of-the-art security analysis tools use to navigate the trade-off between accuracy and computational efficiency for modeling data storage to effectively identify vulnerabilities in smart contracts?
- What factors need to be considered when modeling transient storage in a symbolic execution tool to detect vulnerabilities both accurately and computationally efficiently?
- What implications do the identified factors have for the implementation of the transient storage modeling?
- Which of the identified modeling techniques is the most appropriate for the implementation of the modeling of transient storage in a symbolic execution tool?

1.3 Methodology

1.3.1 Literature Review

In order to determine the current state of data storage modeling techniques in symbolic execution, particularly for the EVM but also regarding conventional techniques, an examination of the literature and of the implementation of publicly available symbolic execution tools is necessary.

1.3.2 Definition of Objectives for the Solution

We will define what the capabilities of our implementation should be. We will also examine what levels of accuracy and computational efficiency are necessary for a symbolic execution tool to discover vulnerabilities in smart contracts and what this implies for modeling of a specific data storage type.

1.3.3 Design and Development

We will explore what factors need to be considered in order to make the best decisions for the implementation of the modeling for transient storage, guaranteeing high computational

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

efficiency as well as accuracy. We will investigate what implications the identified factors have for the design of the modeling of transient storage.
We will extend the data storage modeling of Mythril to account for transient storage and describe in detail our implementation and the rationale behind it.

1.3.4 Demonstration

The Solidity compiler supports transient storage when used in inline assembly. Using this method, we will demonstrate our implementation by analyzing a set of smart contracts that represent different use cases. Thereby, we showcase how our extension of Mythril performs in different scenarios.

1.3.5 Evaluation

We will evaluate our modeling of transient storage by testing it against a set of smart contracts. The accuracy and efficiency of transient storage will be empirically compared against the accuracy and efficiency of the existing modeling of storage, which will act as a baseline.

1.4 Related Work

In 2008, De Moura et al. [DMB08] at Microsoft Research Redmond developed the state of the art satisfiability modulo theories (SMT) solver Z3. Z3 supports several different theories, including arithmetic, bit-vectors, arrays and uninterpreted functions. Z3 has become an industry-standard for symbolic execution. Most symbolic execution tools for smart contracts (including Mythril) generate constraints based on the simulated data storage operations and rely on an efficient SMT solver to solve those constraints, making Z3 very relevant to our research.

In 2011, Godefroid [God11] introduces a novel way of utilizing uninterpreted functions in symbolic execution. Instead of just using uninterpreted functions as an abstraction for complex program behavior, which was common in traditional symbolic execution and oftentimes introduced imprecision, Godefroid also systematically recorded uninterpreted function samples by capturing input-output pairs during runtime. Thereby, he addresses some inherent limitations to traditional symbolic execution tools by ensuring consistency in the symbolic execution process: The same input will yield the same hash output across different points in the analysis, preventing the creation of different unrelated symbols representing the hash outputs of the same input. His method is being applied in several symbolic execution tools to model hash functions., e.g. Manticore [MMH+19], a state-of-the-art symbolic execution tool for smart contracts and ELF binaries.

Shoshitaishvili et al. [SWS+16] present a binary analysis framework along with a corresponding implementation in a tool called angr [ang16], which is publicly available.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Angr implements the index-based data storage technique [CARB12]. Angr has become a standard tool for analyzing binary files for vulnerabilities.

Krupp et al. [KR18] present the tool TeEther, also available publicly [tee18]. The tool introduces a novel way of handling hash values symbolically. TeEther treats fixed-sized data storage elements as fixed-size bitvectors, while variable-length data storage elements are handled using Z3's array theory.

VerX, a symbolic execution tool for smart contracts, developed by Permenev et al. [PDT+20] relies on the reduction of temporal property verification to reachability checking. VerX models hash functions very similarly to Mythril. As modeling hash functions is one of the main issues of handling data storage in symbolic execution of smart contracts, and considering the absence of an accompanying scientific paper for Mythril, examining VerX becomes particularly relevant.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 2

Technical Background

In this section, we lay the groundwork by providing the necessary technical background knowledge, a crucial foundation for comprehending the intricacies of the subject matter.

2.1 SMT Solver and Theories

Satisfiability Modulo Theories (SMT) solvers are foundational tools in symbolic execution and constraint solving. There are numerous widely recognized SMT solvers; examples include Z3 [DMB08], CVC4 [BCD+11] and Boolector [NPWB18]. These solvers have the ability to efficiently determine the satisfiability of logical formulas over various theories, such as bitvectors, arrays, and arithmetic, which differentiates SMT solvers from SAT (boolean satisfisability) solvers. SMT solvers are usually integrated in some form of application for solving constraints generated by the application.

As the different theories are what make SMT solvers able to reason about complex satisfiability problems, we will briefly go into the theories important in the context of symbolic execution of smart contracts:

- **Bitvectors** represent fixed-size sequences of binary digits (0s and 1s). They are often used to model low-level operations in hardware and software. Bitvectors allow for precise control over the number of bits in a variable, making them suitable for certain datatypes in smart contracts (e.g. uint8, uint256). SMT solvers can reason about bitvector constraints efficiently, as they deal with discrete values and support bitwise operations, bit manipulation and binary arithmetic.
- **SMT Arrays** are used to represent array-like datastructures in a logical and efficient manner. This makes them a viable option for modeling EVM data storage types that store data in an address-value like structure. SMT arrays enable the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

modeling and analysis of complex array operations, such as element updates and queries, as well as the handling of respective constraints.

2.2 Symbolic Execution

Symbolic execution is a program analysis technique that has evolved over time. It involves systematically exploring a program's execution paths symbolically, without relying on concrete input values. Instead, it tracks constraints on program variables and computes the possible states the program can reach. This approach allows for the detection of bugs, vulnerabilities, and property violations within software systems by employing SMT solvers to check for the reachability of different program states by solving the previously generated constraints. The underlying concepts of symbolic execution were initially introduced in the late 1960s [Flo67] and further formalized in subsequent years [Kin76]. It has since become a fundamental tool in the fields of formal methods, software verification, and program testing, offering valuable insights into program behavior and correctness.

2.3 Data storage types in smart contracts

The data storage architecture within the Ethereum Virtual Machine (EVM) is quite complex. Various data storage types collectively form the data storage landscape of the EVM and serve diverse and essential roles in the execution of smart contracts.

The following four data storage areas primarily serve the purpose of managing data and computations:

- **Calldata** is a read-only area storing function arguments and data for contract calls.
- **Stack** is used for temporary data storage during contract execution, operates as a stack (LIFO). Every invocation of a smart contract starts with an empty stack, with limit of 1024 entries. Its content is discarded once the contract terminates.
- **Memory** is a temporary and expandable data storage for computations within a contract. Every invocation of a smart contract starts with an empty memory. Its content is discarded once the contract terminates.
- **Storage** is a persistent data storage for contract state variables. Operations cause higher gas costs as reading and writing persistent data requires disk access. The storage is initialized when the contract is created, and persists between transactions. Its content can be either reset programmatically by the contract or is discarded when the contract performs a SELFDESTRUCT operation.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The remaining three data storage areas expand the EVM's capability by enabling output delivery, event recording, and logic execution within smart contracts.

- **Return Data** is used to send data back to the caller after a function execution, allowing contracts to provide information as a response.
- **Logs** are a mechanism for event logging, not direct data storage. They are used to emit structured data for external monitoring and notification.
- **Code** is where the contract's bytecode resides, containing the executable logic of the contract.
- **Callstack** can hold up to 1024 entries and manages nested calls between contracts. The callstack is used to store return addresses and context information. Entries are removed once the associated function call completes.

As the functionality and purpose of transient storage (explained in detail in a dedicated chapter 3) is comparable to memory and storage, we will briefly go into some technicalities that are important for understanding how memory and storage are currently modeled in symbolic execution tools.

2.3.1 EVM Memory

Data stored to EVM memory is discarded after each contract execution (and is thus also not persisted across transactions). Memory is byte-addressable (256-bit long addresses, each pointing to one byte). Memory operations, such as loading (MLOAD) and storing (MSTORE) data, are fundamental for efficient data management within a contract. EVM memory follows a big-endian byte order for data storage (most significant byte is located at the smallest address). Memory allocation in the EVM occurs in 32-byte words and can be conceptualized as using an incrementing pointer, where newly allocated memory is added sequentially, expanding the memory space word by word, 32 bytes at a time (one opcode MSTORE8 also allows for writing a single byte). It is important to note that certain complex data types, including dynamically sized arrays and mappings, cannot be stored directly in memory.

2.3.2 EVM Storage

Unlike memory, which is temporary, storage in Ethereum provides a persistent data storage solution and is word-addressable, with each address being 256-bit long and pointing to 256-bit values (one word). Smart contracts use storage to maintain long-term data, such as contract state variables. In terms of data storage format, Ethereum storage is big-endian, same as memory. Unlike memory, which is a linear array, storage is inherently a mapping of 256-bit keys to 256-bit values. This key-value pair structure explains why certain complex data types, such as mappings (in Solidity), are implemented in storage

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

rather than memory. Fixed-size variables are stored in consecutive slots, following their declaration order, while dynamic-size data types utilize a hash-based allocation scheme.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 3

EIP-1153: Transient Storage

This chapter delves into Transient Storage as introduced by Ethereum Improvement Proposal (EIP) 1153. The EIP was created in 2018 [AS18] and as of 15.01.2024 is undergoing peer-review in the Ethereum community. EIP-1153 is strongly anticipated to be a part of the next Ethereum upgrade, known as the "Cancun Upgrade", expected in early 2024.
A great part of this chapter leans on what is specified in the EIP ([AS18]).

3.1 Motivation of Transient Storage

An Ethereum transaction can generate multiple frames of a contract. This can happen when a contract uses the *CALL*, *DELEGATECALL*, *CALLCODE* or *STATICCALL* instruction to either call another contract or itself recursively. In this case, a new execution frame is created and managed by the EVM and further execution happens in the new contract frame. This new frame has its own memory, stack and program counter. As a result, communication between multiple frames of the same contract is currently only possible using one of two options:

- Storage. This comes with high gas costs, as it requires disk access.
- Inputs passed via *CALL* instructions. This is insecure, if there exists an intermediate frame belonging to an untrusted contract.

Several use cases however require inter-frame communication.
Previous attempts addressing this issue have inherent limitations. [Tan19] proposes to reprice SSTORE and SLOAD opcodes to make it cheaper for use cases where the lifetime of the data is limited by the end of the transaction. This is not a satifying solution as the maximum refund is limited to only 20% of the transaction gas cost.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

'EIP-1153: Transient Storage' offers inter-frame communication without security concerns and using significantly less gas than storage as transient storage does not require disk access.

3.2 Technical Details of EIP-1153

The functionality of transient storage is virtually equivalent to storage, with the important difference that it is discarded after every transaction. Storage in contrast persists between transactions and hence requires disk access.

EIP-1153 specifically introduces "Opcodes for manipulating state that behaves identically to storage but is discarded after every transaction". The opcodes being added are *TSTORE* and *TLOAD* for writing and reading, respectively, which are similar to *SSTORE* and *SLOAD*. Addressing (32-byte addresses pointing to 32-byte values) in transient storage also works equivalently to storage.
Gas costs for *TSTORE* and *TLOAD* are significantly lower (100 gas each) than for the storage opcodes (100-20000 gas [Woo23], depending on whether the storage slot has been read/written to yet).

3.3 Use Cases and Applications

As everything that is possible with transient storage would also be possible with conventional storage, transient storage does not directly provide new functionality. However, transient storage is useful in cases where inter-frame communication is necessary, but the data stored does not need to persist between transactions.

This can be useful for a range of use cases and applications. [AS18] lists the following ones:

- **Reentrancy Locks.** A security mechanism designed to prevent recursive calls of functions when interacting with untrusted contracts.
- **On-Chain Computable CREATE2 Addresses.** A use case where constructor arguments are read from the factory contract's state instead of being passed as part of the initialization code hash.
- **Single Transaction ERC-20 Approvals.** Temporary ERC-20 token approvals within a single transaction.
- **Fee-on-Transfer Contracts.** A fee to be paid in order to unlock temporary transfers within a transaction.
- **Till Pattern.** Enable users to perform multiple actions in a single transaction, with a check at the end to ensure the "till" is balanced.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- **Proxy Call Metadata.** Storing additional metadata temporarily for implementation contracts, especially useful for immutable proxy constructor arguments.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 4

Comparative Analysis of existing Data Storage Modeling Techniques

Numerous analysis tools have been developed for addressing the issue of finding vulnerabilities in smart contracts, some of which employ symbolic execution. During symbolic execution, these tools need to model the different data storage types of the EVM as well as the operations performed on them. In order to get a solid understanding for existing data storage modeling techniques in symbolic execution tools for smart contracts, we conducted a review of the code of several of the most well-known and respected open-source tools. In the following, the different methods we extracted are presented, explained and compared.

4.1 Analysis of the tools

Di Angelo and Salzer present a detailed overview of many analysis tools for smart contracts [DAS19]. We apply similar criteria for choosing the smart contract analysis tools from which we extract the data storage modeling techniques. The tool needs to:

- perform symbolic execution
- aim at finding security vulnerabilities
- be open-source

We also consider whether the tool:

- is recently maintained and well-structured
- is reasonably well-respected in industry and/or in the literature

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Finally, out of nine security analysis tools for smart contracts that employ symbolic execution identified by [DAS19], our choice fell on the tools *Mythril*, *TeEther* and *Manticore*. The other tools have been disregarded due to the fact that they are either stale, have not been publicly released or extend another tool (reusing the symbolic execution engine and its data storage modeling).

For this comparative analysis, we will focus on the techniques for modeling smart contract *storage* and *memory*, as these two memory areas are most closely related to transient storage.

4.1.1 Mythril

Mythril [Con18] being the tool we extend with a model for transient storage, an analysis of the existing data storage modeling techniques applied in this tool are obviously very relevant. Mythril is a symbolic execution security analysis tool for smart contract bytecode. It supports multi-transactional exploits. Mythril has been published on Github in 2018 and is still actively maintained by the company ConsenSys.

- **Storage** is modeled as an SMT array (theory of arrays). The indices and values are modeled as 256-bit bitvectors.
- **Memory** is modeled as a Python dictionary. The indices are modeled as 256-bit bitvectors, the values are modeled as 8-bit bitvectors.

4.1.2 TeEther

TeEther [tee18] (accompanied by a paper [KR18]) is a symbolic execution tool for finding vulnerabilities and generating exploits for smart contracts, focussing on exploits that transfer Ether to an arbitrary address. It supports multi-transactional exploits. The TeEther paper has been published in 2018, the public release of the tool followed in 2019.

- **Storage** is modeled as an SMT array (theory of arrays). The indices and values are modeled as 256-bit bitvectors.
- **Memory** is modeled as an SMT array (theory of arrays). The indices are modeled as 256-bit bitvectors, the values are modeled as 8-bit bitvectors.

4.1.3 Manticore

Manticore [man19] (accompanied by a paper [MMH+19]) is an analysis tool that supports symbolic execution for both traditional ELF binaries as well as for EVM bytecode.

- **Storage** is modeled as an SMT array (theory of arrays). The indices and values are modeled as 256-bit bitvectors.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- **Memory** is modeled as an SMT array (theory of arrays). The indices are modeled as 256-bit bitvectors, the values are modeled as 8-bit bitvectors.

4.2 Analysis and Comparison of Methods

In the following, we juxtapose the techniques with which storage and memory are modeled by the different tools.

4.2.1 Modeling Smart Contract Storage

All three tools model storage as an SMT array with appropriately sized 256-bit bitvectors as indices and values (2^{256} addresses, each addressing one word, i.e. 32 bytes or 256 bits). This model perfectly reflects how the EVM works and can accurately reason about symbolic reads and writes on the SMT array representing storage. The only deviation from precise modeling is related to the SHA3 hash function, which is used in high-level languages like Solidity for certain data types. This hash-based allocation scheme is a feature of these languages, not of EVM storage itself. In the EVM context, storage is simply a mapping of 256-bit keys to 256-bit values. Hash functions are modeled as uninterpreted functions, which affects the precision of reasoning about storage indices derived from hashes.

4.2.2 Modeling Smart Contract Memory

When it comes to memory, TeEther and Manticore are consistent in their approach of using an SMT array. For memory, the tools capture the fact that memory is byte-addressable with 256-bit bitvectors as indices and 8-bit bitvectors as values (2^{256} addresses, each addressing one byte, i.e. 8 bits).
Mythril handles memory differently, though: A simple Python dictionary is used. This dictionary also uses bitvectors to accurately capture the size of addresses/bytes as index/value pairs.

4.2.3 Using an SMT Array vs. using a Dictionary

The analysis of data storage modeling techniques has come up with two methods for modeling an index-value pair-based datastructure: SMT arrays and dictionaries.

Conceptually, a dictionary works the same as an SMT array, providing the possibility to perform read and write operations on a structure of addresses pointing to values. However, in the context of symbolic execution, SMT arrays provide more accuracy than dictionaries. This is due to the fact that SMT arrays can reason about the SMT constraints put on the bitvectors that are used as indices. An SMT array recognizes whenever a bitvector used as index is potentially equivalent to an already existing index, even when the two operations (the one writing the value and the other one retrieving the value) use different

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

symbols as index to access the SMT array. Reasoning about the SMT constraints of the symbols used as indices, however, makes SMT arrays less efficient.

A Python dictionary is a much more light-weight data structure than an SMT array and due to its simpler nature and its inability to reason about SMT constraints, it is more efficient. However, a Python dictionary comes with the downside of being inaccurate in cases of symbolic storage/memory access. A dictionary will treat different symbols as different indices, even if the symbols are potentially equivalent according to their SMT constraints: Writing a value to transient storage and later attempting to retrieve that value with a symbolic index (potentially equivalent to the written index) will fail, as the dictionary treats any symbol as separate index.

Whether sacrificing accuracy for efficiency using a dictionary is acceptable therefore depends on two things:

1. Are symbolic storage/memory accesses with different but equivalent symbols common? Are they rare enough so that the inaccuracies can be disregarded?

2. How much more efficient is the dictionary in comparison to an SMT array? Is the increased efficiency substantial enough to justify the inferior accuracy?

Using a dictionary for modeling storage would result in numerous false positives/negatives during a security analysis. This is due to the fact that symbolic storage accesses with different but equivalent symbols occur quite often, since the purpose of storage is long term, as storage is part of the blockchain's state and persists across transactions. For example: storing a value within one transaction and reading it in a later transaction is quite common, but will result in two different (but equivalent) symbols used as indices. Therefore, we conclude that using a dictionary for modeling smart contract storage is not an option, as the number of false positives/negatives would render a security analysis useless.

Using a dictionary for modeling memory, however, works quite well. As memory does not persist across transactions and is reset after a transaction concludes, smart contracts mainly use memory as short term cache and symbolic memory accesses with different but equivalent symbols are therefore rare. Writing a value at a symbolic index (e.g. using msg.sender as key to a mapping) in one transaction and reading it in a later transaction is a common case of equivalent symbolic indices, which are interpreted as different symbols by the symbolic execution engine, since two different transactions are involved. This is not a common case for memory, however, as memory only persists per transaction. Whether such a case of symbolic memory access can still occur, obviously depends on the smart contract being analyzed, but there are few use cases for it. As a result, Mythril sacrifices accuracy for efficiency by modeling memory as dictionary and not as SMT array.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The modeling remains quite accurate, since the cases introducing inaccuracies are rare.

We can therefore conclude that the choice between using an SMT array or a dictionary for modeling a smart contract data storage type depends on whether the use cases typically result in different but equivalent symbolic indices for the data storage type in question, which in turn depends on whether the data storage type is used for persistent storage or for per-transaction temporary storage.

4.3 Additional observations

Modeling Hashes. Hashes, particularly the keccak256 (a variant of SHA-3) function used by Ethereum, are significant in smart contracts, serving various purposes in both code and system design. In addition to their direct application within contract code, hashes are integral to high-level languages like Solidity for storage allocation mechanisms, determining the location of mappings and dynamically sized arrays. However, if the keccak256 function were to be modeled with precise SMT constraints in a symbolic execution engine, it would significantly hinder the engine's performance due to the computational complexity and resource-intensive nature of accurately simulating cryptographic hash functions. Mythril models hash functions as uninterpreted functions, which is a common technique in symbolic execution. The input and output data of the hash function are modeled as 256-bit bitvectors. SMT constraints are applied to the output hash to mimic properties of EVM hash functions in order to enable handling hashes symbolically.
Mythril and Manticore capture input-output pairs of hash functions during runtime, a technique introduced by Godefroid [God11]. This is relevant for modeling a data storage type like transient storage, especially if a dictionary is used. The reason for this is that whenever the keccak256 function is encountered by the symbolic execution engine, it normally creates a new bitvector. If no input-output pairs would be captured and the same input symbol would be hashed twice, this would result in two different output symbols. This would obviously lead to problems when a data storage type that uses hash-based allocation scheme (for e.g. mappings) is modeled as a dictionary, because multiple accesses to the same mapping slot (calculated by one input symbol) would result in different output symbols and would hence be recognized as accesses to different mapping slots by the symbolic execution engine.

EVM Stack. All three tools extend Python's list class for modeling the EVM stack. This might be due to Python's list class being similar in nature to the EVM stack, offering push and pop methods with a Last-In-First-Out functionality. This enables repurposing Python's list class to model the EVM stack accurately and intuitively.

SMT Solver. All three tools employ Z3 as the SMT solver to solve the constraints generated by the symbolic execution engine.

Bitvector Theory. All three tools model all symbolic values using the SMT theory of bitvectors, providing a low-level, fine-grained representation and allowing for detailed

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

operations at the bit-level. The bitvector theory aligns well with the fixed-size data types (like 256-bit integers) in Ethereum. No abstraction is applied by the tools to symbolic values by modeling them in a more coarse-grained manner, e.g. using integer theory.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 5

Definition of Requirements

In the following, we define the requirements that an implementation of transient storage in a symbolic execution security analysis tool for smart contracts should fulfill. We also explain the role that the modeling of transient storage plays in the definition of requirements.

5.1 Modeling of storage as baseline

Transient storage is defined as “state that behaves identically to storage, except that transient storage is discarded after every transaction” [AS18]. The fact that the behavior within one transaction is identical qualifies the modeling of storage as comparison / benchmark candidate for the modeling of transient storage in symbolic execution tools.

5.2 Requirements towards accuracy

We are extending a security analysis tool that detects vulnerabilities in smart contracts. The utility of a security analysis tool depends to a large degree on correctly identifying security issues. Inaccurate modeling of data storage worsens the accuracy of identified vulnerabilities. Consequently, an important metric for evaluating accuracy is the number of true/false positives/negatives. However, if the evaluation results contain false positives/negatives, it is hard to tell to what degree these mistakes can be attributed to our extension, which only consists of the modeling of one data storage type; false positives/negatives can also result from a faulty symbolic execution engine for example.

For this reason, we rely on the modeling of storage that is already present in Mythril as a baseline.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

We define as a requirement that our model of transient storage is as accurate as the model of storage in the state-of-the-art. The details regarding the comparison of accuracy in the context of an empirical evaluation are described in chapter 8.

5.3 Requirements towards efficiency

Efficient modeling of EVM operations by smart contract symbolic execution tools is crucial to maintain the efficiency of smart contract analysis. Inefficient modeling can significantly increase the computational complexity and runtime of the analysis, leading to substantial time delays.

As it is difficult to evaluate efficiency objectively without comparison, we also rely on the modeling of storage, that is already present in Mythril, as a baseline. Storage in Myhtril (and in other state-of-the-art tools) is modeled as an SMT array, which is the less efficient option compared to using a simple dictionary (as explained in chapter 4).

Consequently, we define as a requirement that the operations on the transient storage model are at least as efficient as the operations on the storage model. The details regarding the comparison of efficiency are described in chapter 8.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 6

Development: Modeling Transient Storage

In the previous chapter 5, we defined the state-of-the-art modeling techniques for modeling smart contract storage as the baseline for transient storage modeling.

We also defined in chapter 4 that the modeling of storage is almost completely precise (compared to how storage works in the EVM), with the exception of the hash-based allocation scheme that is used for some data structures stored to storage. The imprecision of modeling hash functions is a well-known problem in symbolic execution and is an issue separate from modeling storage. Therefore, we deem the modeling of storage itself as being as precise as it can be.

The comparative analysis in chapter 4 has shown that another option for modeling transient storage besides using an SMT array is using a dictionary. This is an interesting alternative for modeling transient storage because the inaccuracies, introduced by the dictionary's inability to reason about SMT constraints, are relevant especially for data storage types that store data long term and therefore experience more data storage accesses with (different but potentially equivalent) symbolic indices.

Transient storage is only used for short term, per-transaction purposes, and differs in precisely this way from storage, making it a candidate for modeling it using a simple dictionary. This can lead to gains in efficiency. Therefore, we use a dictionary to model transient storage, which allows us to evaluate whether it is in fact a viable option.

Our implementation is published on GitHub [Wey23].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

6.1 Extension of Mythril

We focus on extending an existing tool, Mythril, with a transient storage model. Our extension is based on Mythril version 0.24.5, specifically commit `b8ad3a9`, pushed on Jan 16, 2024.

6.2 Development Environment and Tools

Mythril is written in Python. We also use Python for our extension, which is necessary as our extension is implemented directly inside the existing code.

We use Github for version control and better overview of the design process.

6.3 Design Choices and Implementation Details

As discussed at the start of this chapter, we implement transient storage as a dictionary in order to determine if this approach allows for efficiency gains without introducing inaccuracies. The indices and values of the dictionary are modeled as 256-bit bitvectors.

6.3.1 Mythril Architecture

In order to explain how the dictionary representing transient storage is integrated into Mythril, we take a brief look at the existing implementation and architecture of Mythril.

In the context of symbolic execution, Mythril keeps track of different states of smart contracts and their execution. In this regard, Mythril implements multiple Python classes that are structured in a hierarchy-like structure. This structure needs to be considered when deciding how to extend Mythril with transient storage, as transient storage needs to be placed in the proper place inside this structure in order to properly simulate the EVM.

We start by giving a brief overview of the most important Python classes situated in this hierarchy, along with the information we deem most important to understand the role of each class in symbolic execution and in managing data storage. Some of the following descriptions are derived from Mythril's documentation [Con18].

GlobalState class is at the top of the hierarchical structure. Mythril's symbolic execution engine can keep track of a list of different GlobalStates. GlobalState keeps track of a WorldState, an Evironment and a MachineState. It also manages a transaction stack (of all transactions leading to that specific global state) as well as a list of annotations, which are used to note detected issues.

WorldState class represents the world state as described in the Ethereum Yellow Paper. It keeps track of different accounts and their balances.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Environment class represents the current execution environment for the symbolic execution engine. It keeps track of an active account as well as several values relevant to the transaction (origin, active_account, ...).

Account class represents Ethereum accounts and keeps track of the account's code and the account's storage.

Storage class represents smart contract storage and keeps track of the data stored to the persistent storage of a smart contract.

MachineState keeps track of contract frame specific values, like Memory, the Stack and the program counter.

Memory represents smart contract memory and keeps track of the data stored to the ephemeral memory of a smart contract.

Stack represents the smart contract stack and keeps track of the data pushed on and popped off the stack.

6.3.2 Transient Storage in Mythril

In the following, we list and briefly describe the different components and changes to the existing code that make up our extension of Mythril. Most of these are situated in Mythril's `mythril/laser/ethereum/state` directory, as it contains all existing modeling of state and data storage (essentially the classes described above in 6.3.1).

- Transient storage itself (the dictionary and all relevant methods for interacting with and copying it) are implemented in a separate file `mythril/laser/ethereum/state/transient_storage.py`.
- Transient storage is integrated in the Account class (same as storage). The reason for this decision is that the Account class is part of the Environment class. Transient storage fits well into this, because the Environment of an account persists across the entire transaction and represents the larger context of the transaction and account-specific information that might be relevant across multiple computational steps. The other option would have been to integrate transient storage in the MachineState (same as memory). However, as mentioned above in 6.3.1, MachineState represents the execution frame specific values. As transient storage persists across execution frames, it would not fit into Mythril's MachineState class.
- The functionality of the transient storage opcodes `TLOAD` and `TSTORE` is implemented in `mythril/laser/ethereum/instructions.py`
- Adding the transient storage opcodes to `mythril/support/opcodes.py` ensures that Mythril can recognize the opcodes when disassembling bytecode.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Figure 6.1 depicts the hierarchy of the existing classes in Mythril as well as a transient storage class added in the context of our extension.

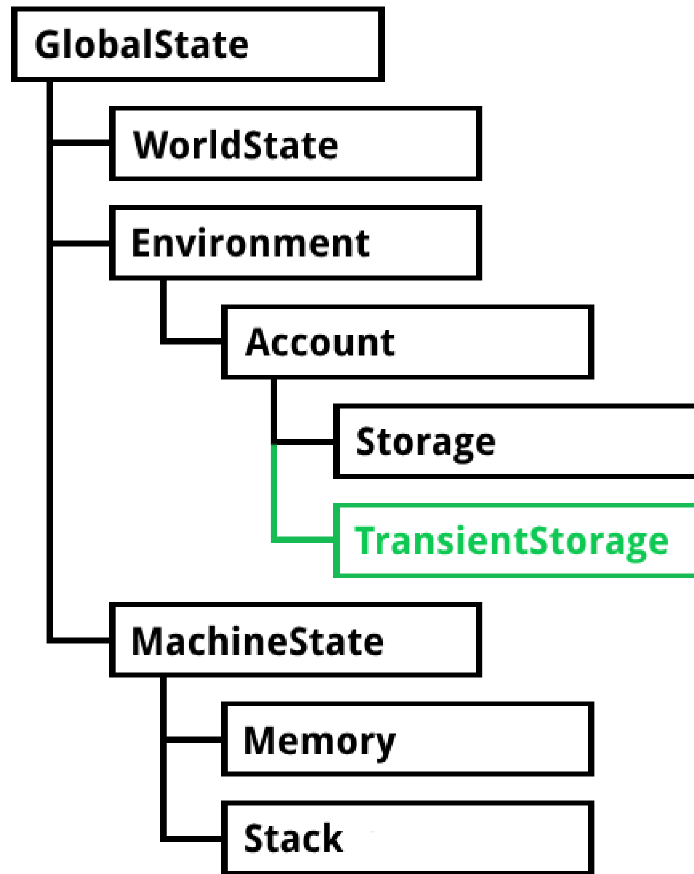

Figure 6.1: Data Storage in Mythril

6.3.3 Transient Storage in the EVM

Figure 6.2 depicts an interpretation of the storage structure of the Ethereum virtual machine including transient storage. This visualization was uploaded to the Fellowship of Ethereum Magicians forum.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

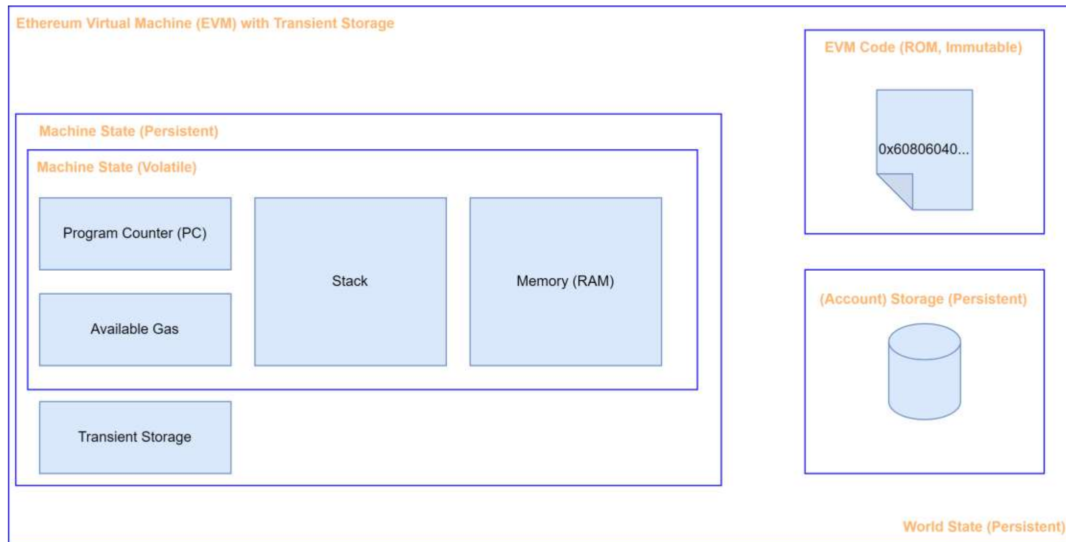

Figure 6.2: Adjusted EVM With Transient Storage [Pas]

This visualization was uploaded by an independent user to the Ethereum Magicians forum (a platform dedicated to discussions mainly about EIPs) and is not officially recognized by the authors of EIP-1153. However, it is worth examining it more closely in relation to our model of transient storage.

In this visualization, transient storage is more closely associated with the MachineState. The figure differentiates between a volatile machine state (which contains the execution frame specific data and is equivalent to the MachineState class in Mythril) and a persistent machine state which exclusively contains transient storage. The storage is separate from what the author of this visualization calls "persistent machine state". The author of this visualization uses the concept of this persistent machine state to explain the fact that the persistence of transient storage is to be understood somewhere between the ephemeral nature of the EVM's volatile machine state and the EVM's persistent storage.

Our implementation of transient storage associates transient storage less closely with the volatile machine state and instead places it into the Environment class. We believe this is appropriate, because Mythril's architecture (specifically the environment class) is structured differently than what the visualization presents. The environment class (even though it contains storage) does not represent a cross-transaction persistent state and is therefore ideal to also keep track of account data that should be persisted across one whole transaction, but not across multiple transactions (which is exactly what is needed for transient storage).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

6.4 Mythril Analysis Modules

We modify some analysis modules so that the analysis recognizes potential vulnerabilities relevant to transient storage.

- The `Arbitrary Write` module now supports recognition of arbitrary write to transient storage. This case would occur when untrusted user input is used to determine the transient storage slot to write to in inline assembly.
- The `State Access after external Call` module now supports recognition of transient storage access after external call. Additionally, the name has been changed to `Storage / Transient Storage Access after external Call`. Even though transient storage is not part of the state, transient storage is persisted across function calls / execution frames, making it necessary to regard accesses after external function calls as potential vulnerability.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

CHAPTER 7

Demonstration

In the following, we demonstrate the application of our extension of Mythril [Wey23]. We aim at showing its capability in modeling transient storage according to how transient storage is defined in EIP-1153, in order to identify vulnerabilities involving transient storage in smart contracts.

7.1 Setup

As defined in chapter 6, our extension of Mythril consists of the capability of recognizing transient storage opcodes in smart contract bytecode and modeling them appropriately. In order to demonstrate our extension, we therefore need to prepare appropriate smart contracts' bytecode, which involves transient storage in different scenarios & use cases.

7.1.1 Solidity Compiler

On Ethereum's main chain, transient storage was activated with the Cancun fork in March 2024. The Solidity compiler supports the opcodes for transient storage in inline assembly since version 0.8.24, released on Jan 26, 2024. Consequently, there are two methods to obtain a contract's bytecode that includes transient storage: Either (a) compose a contract using transient storage opcodes in assembly and compile it using the latest version of the solidity compiler or (b) compose a contract using storage, compile it and then replace the storage opcodes with transient storage opcodes in the bytecode. Both methods yield equivalent bytecode (assuming the inline assembly is coded properly).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

7.1.2 Construction of Smart Contracts

The smart contracts we construct involve different scenarios using transient storage. They include vulnerabilities that, when analyzed with our extension of Mythril, demonstrate the reliability of our modeling approach.

The contracts are written in Solidity, with all transient storage operations defined in inline assembly.

The scenarios in the smart contracts we compose contain code that sends the contracts balance to the caller of the contract. The conditions of whether this part in the code is reached depend on the previous interactions with transient storage. We can use Mythril's "Unprotected Ether Withdrawal" analysis module to verify reachability. This allows demonstrating the correct functioning of transient storage.

7.2 Executing the Tool

As transient storage only persists across one transaction and is discarded afterwards, we execute the analysis with the number of transactions limited to 1. However, we also demonstrate separately that our extension of Mythril, when executed using multiple transactions, does not report persist data in transient storage across transactions and consequently does not indicate recognized vulnerabilities that should only be reachable with data persisted across transactions.

In order to receive the bytecode, we compile the smart contracts using the following command:

```
./solc␣--bin␣PATH_TO_SOL_FILE␣--evm-version␣cancun
```

We run our extension of Mythril on the demonstration contracts using the following command:

```
./myth␣analyze␣-c␣"BYTECODE_OF_CONTRACT"␣-t␣1
```

The `-t` flag indicates the limit for the number of transactions to use. This is set to 1 as transient storage only persists across 1 transaction. The non-persistence of transient storage in our extension of Mythril is demonstrated as well in one of the demonstration contracts, which is analyzed without this limit of one transaction.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

7.3 Demonstration Contracts

In this section, we discuss the demonstration contracts together with the analysis results. We explain how the results demonstrate proper functionality of our transient storage model.

7.3.1 Simple Case

The contract in Listing 7.1 below contains a vulnerability (Ether can be withdrawn if the correct input value is given) requiring single transaction. The control flow involves an internal function call. Running our extension of Mythril on this contract detects an "External Call to User-Supplied Address" as well as an "Unprotected Ether Withdrawal" vulnerability, which is expected. The transaction sequence is given as a call to the `storeAndWithdraw` with input 42.

```
pragma solidity ^0.8.24;

// This contract demonstrates transient storage in a simple case
    with functions calls inside the same smart contract, creating
    multiple execution frames of the contract.

contract SimpleCase {

    constructor() payable {}

    function storeAndWithdraw(uint256 _input) public {
        assembly {
            tstore(0x123, _input)
        }

        this.withdrawBalance();
    }

    function withdrawBalance() public {
        uint256 storedValue;
        assembly {
            storedValue := tload(0x123)
        }

        require(storedValue == 42, "Stored value is not equal to
            42.");

        (bool sent, ) = msg.sender.call{value:
            address(this).balance}("");
        require(sent, "Failed to send Ether");
    }
}
```

Listing 7.1: Simple Case

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

7.3.2 No Persistence across Multiple Transactions

The contract in Listing 7.2 below demonstrates that transient storage is not persisted across transactions. It contains a vulnerability reachable only if transient storage was persisted across multiple transactions. Running our extension of Mythril on this contract detects no vulnerabilities, as expected. Using storage instead of transient storage (replacing `TLOAD` and `TSTORE` with `SLOAD` and `SSTORE` respectively) detects a vulnerability.

Note that this contract is analyzed without the limit of one transaction. Without a specified limit, Mythril defaults to a limit of two transactions, which is enough to trigger the vulnerability, if transient storage was persisted across transactions.

```
1  pragma solidity ^0.8.24;
2
3  // This contract demonstrates that transient storage does not
       persist across multiple transactions.
4
5  contract NoPersistenceAcrossTransactions {
6
7      constructor() payable {}
8
9      function store(uint256 _input) public {
10         assembly {
11             tstore(0x80, _input)
12         }
13     }
14
15     function withdrawBalance() public {
16         uint256 storedValue;
17         assembly {
18             storedValue := tload(0x80)
19         }
20
21         require(storedValue == 123, "Stored value is not equal to
               123.");
22
23         (bool sent, ) = msg.sender.call{value:
               address(this).balance}("");
24         require(sent, "Failed to send Ether");
25     }
26 }
```

Listing 7.2: No Persistence across Multiple Transactions

7.3.3 Failure to recognize the potential equivalence of different symbolic indices

The contract in Listing 7.3 below demonstrates that transient storage as we implement it does not recognize the potential equivalence of two different symbolic indices. The

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

reason for this is that we implement transient storage using a dictionary. The failure to recognize potentially equivalent, but different symbolic indices is therefore expected. The function symbolicIndices below takes two user parameters (leading to different, but potentially equivalent symbols during symbolic analysis) which are used as indices to transient storage. As expected, the analysis fails to recognize that the contract is vulnerable to Unprotected Ether Withdrawal and External Call to User-Supplied Address. Even though symbolic access to a data storage type that only persists for one transaction is rare, it is important to demonstrate this expected shortcoming of our implementation.

```
1  pragma solidity ^0.8.24;
2
3  // This contract demonstrates that our modeling of transient storage
       fails to recognize the potential equivalene of two different
       symbolic indices.
4
5  contract EquivalentSymbolicIndices {
6
7      constructor() payable {}
8
9      function symbolicIndices(uint256 _key1, uint256 _key2) public {
10         assembly {
11             tstore(_key1, 123)
12
13             // if transient storage at key2 is equal to what has
                   been stored at key1, send balance of contract to
                   msg.sender
14             if eq(tload(_key2), 123) {
15                 let success := call(gas(), caller(), selfbalance(),
                       0, 0, 0, 0)
16             }
17         }
18     }
19 }
```

Listing 7.3: Potentially Equivalent Symbolic Indices

7.3.4 Hashes as Indices

The contract in Listing 7.4 below demonstrates that our modeling of transient storage does recognize two different, but equivalent hashes as indices. Our extension of Mythril detects an Unprotected Ether Withdrawal as well as an External Call to User-Supplied Address.

As Mythril models hashes using Bitvectors (meaning symbols), when the same value is hashed twice, this should result in two different symbols. Therefore, one might expect the same behaviour as above, meaning that symbolic execution would not recognize the equivalence of the two hashes, even if the hashes result from the same input symbol. However, this is not the case. The reason for this is that Mythril captures input-output

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

pairs of hashes (Godefroid's method [God11]). The fact that our extension does recognize this is not directly a quality of our extension, but instead a feature of Mythril's existing handling of hashes. However, it is relevant to demonstrate this in the context of our implementation of transient storage, as hashes play a central role when indexing in storage/transient storage.

```
1  pragma solidity ^0.8.24;
2
3  // This contract demonstrates that our modeling of transient storage
      does recognize the equivalence of two hashes used as indices.
4
5  contract HashesUsedAsIndices {
6
7      constructor() payable {}
8
9      function hashStoreAndWithdraw(uint256 _key1) public {
10         // Compute the hash of _key1 and _key2 to use as indices
11         uint256 hashIndex1 =
               uint256(keccak256(abi.encodePacked(_key1)));
12         uint256 hashIndex2 =
               uint256(keccak256(abi.encodePacked(_key1)));
13
14         assembly {
15             // Store 123 at the computed index
16             tstore(hashIndex1, 123)
17
18             // if transient storage at key2 is equal to what has
                   been stored at key1, send balance of contract to
                   msg.sender
19             if eq(tload(hashIndex2), 123) {
20                 let success := call(gas(), caller(), selfbalance(),
                       0, 0, 0, 0)
21             }
22         }
23     }
24 }
```

Listing 7.4: HashesAsIndices

7.3.5 Arbitrary Write to Transient Storage

The contract in Listing 7.5 demonstrates the `Arbitrary Write to Storage / Transient Storage` detection module. The vulnerability is enabled by writing to transient storage with a user-supplied transient storage slot number. The vulnerability is detected as expected.

```
1  pragma solidity ^0.8.24;
2
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
// This contract demonstrates the Arbitrary Write to Transient
   Storage detection module.

contract ArbitraryWrite {

    function arbitraryWrite(uint256 _key, uint256 _value) public {
        assembly {
            tstore(_key, _value)
        }
    }
}
```

Listing 7.5: Arbitrary Write to Transient Storage

7.3.6 Access to Transient Storage after External Call

The contract in Listing 7.6 demonstrates the `Access to Storage / Transient Storage after External Call` detection module. This vulnerability is triggered by writing to transient storage after a call to an external contract. The vulnerability is detected as expected.

```
pragma solidity ^0.8.24;

// This contract demonstrates the Transient Storage Access after
   external Call detection module.

contract AccessAfterExternalCall {

    function accessAfterExternalCall() public {

        msg.sender.call("");

        assembly {
            tstore(0x123, 0x456)
        }
    }
}
```

Listing 7.6: Access to Transient Storage after External Call

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 8

Evaluation

We evaluate our extension of Mythril in terms of accuracy and efficiency.

As stated in the requirements in chapter 5, we use the modeling of storage as a baseline to evaluate our modeling of transient storage. We rely on the fact that in the EVM, transient storage and storage behave identically if the number of transactions is limited to 1. This is especially relevant for comparing accuracy, because an analysis for one transaction should detect the same vulnerabilities for a contract, regardless of whether that contract uses storage or transient storage for its global variables. For this reason, we evaluate our implementation using two versions of a set of contracts: Once the contracts use storage for global variables and once they use transient storage for global variables. Other than that, the two versions are identical.

8.1 Contracts used for evaluation

The evaluation was conducted using the set of “Solidity example” contracts published alongside the original Mythril tool [Con18]. This is a set of 14 Solidity files, mostly containing a single contract while a few files contain multiple contracts. The contracts illustrate different use cases and contain various vulnerabilities detectable by Mythril’s detection modules (this also allows us to check whether our extension properly works in conjunction with these detection modules). The contracts originally use storage for the global variables.

8.2 Preparation of evaluation contracts

We need two versions of each contract: One using storage (which already exists) and one using transient storage.

TU Bibliothek Your knowledge hub WIEN

We compile the contracts with the Solidity compiler version 0.8.24. In some cases, the contracts use deprecated syntax (like missing function modifiers `virtual` and `override`, usage of `now` instead of `block.timestamp`, certain forms of type casting, etc.), which we update using the respective up-to-date Solidity features.

The compilation results in bytecode containing opcodes for storage interactions (SLOAD and SSTORE). We create copies of each contract's bytecode and replace the storage opcodes with their respective transient storage pendant (TLOAD and TSTORE).

8.3 Accuracy

We analyze both versions of each contract using our extension of Mythril. The command-line output is published alongside our extension on GitHub. We compare the identified vulnerabilities for both versions of each contracts.

For the contracts in 12 of the 14 Solidity files, the same vulnerabilities are identified for both versions of each contract. For the remaining two contracts (`rubixi.sol` and `weak_random.sol`), `Integer Arithmetic Bugs` are identified as vulnerability by the analysis of the version using storage, while these vulnerabilities are not recognized by the analysis of the version using transient storage.

To understand the reasons for this discrepancy, we use the information provided by Mythril for each identified vulnerability, in particular the function name and the exact bytecode address where the issue was found. This allows for further inspection, which we conduct using the Remix IDE [Rem]. Running the transactions leading to the identified vulnerabilities, we find that these integer arithmetic bugs (which essentially represent a potential underflow or overflow) result not from vulnerabilities in the contract, but from low-level EVM computations involving a `SUB` opcode. One example that could lead to such a case is the equality check for two values read from storage. The EVM performs this equality check by subtracting the second value from the first value and checking whether the result is 0. This subtraction (`SUB` opcode) is what causes a potential underflow, if the second value is larger than the first. Hence, these vulnerabilities identified by Mythril (for the version using storage) are false positives, as the code does not actually lead to an underflow or overflow relevant to the logic of the contract.

We therefore deem that our modeling of transient storage fulfills the baseline requirement towards accuracy defined in chapter 5, since the identified vulnerabilities are identical to the ones found in the contracts using storage (apart from the two false positives).

8.4 Efficiency

In order to evaluate the computational efficiency of our implementation of transient storage modeling, we execute the tool ten times on the contract's version using storage and 10 time for the contract's version using transient storage. We use Linux's 'time'

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

command, which measures the total elapsed time for the tool to complete. We then calculate the mean execution time as well as the standard deviation for the 10 runs.

Mythril's symbolic execution engine features a plugin designed to profile the performance of the different EVM instructions as modeled by symbolic execution. The plugin records opcode-specific execution times, which are used to generate statistics for each opcode, including absolute and percentage (relative to the whole analysis) execution times.

We utilize these statistics to compare the efficiency of the modeling of transient storage opcodes to the baseline (the existing modeling of storage opcodes).

Contract	Storage		Transient Storage		Difference [%]
	Mean [s]	σ [s]	Mean [s]	σ [s]	
BECToken1	6.7424	0.1428	6.5751	0.1572	- 2.48%
BECToken2	358.7670	60.5354	151.1230	11.7600	- 57.88%
BECToken3	4.5255	0.0917	4.4438	0.1204	- 1.81%
BECToken4	5.7368	0.1942	5.6689	0.0949	- 1.18%
BECToken5	126.2600	7.4414	124.0750	9.3496	- 1.73%
BECToken6*	2.7902	0.5081	2.7926	0.5270	+ 0.09%
BECToken7	13.9255	0.1811	12.9056	0.2663	- 7.32%
WalletLibrary1*	2.7689	0.4640	2.7540	0.4886	- 0.54%
WalletLibrary2	113.8440	0.9489	77.5613	1.4764	- 31.87%
calls	26.3215	1.6657	25.1456	1.5213	- 4.47%
etherstore	18.5326	0.5547	12.8863	0.2460	- 30.47%
hashforether*	3.7008	0.0970	3.6428	0.0906	- 1.57%
killbilly	5.7143	0.1025	5.6821	0.1607	- 0.56%
origin	5.4522	0.6025	5.3856	0.6455	- 1.22%
returnvalue	5.7183	0.7833	5.5538	0.6944	- 2.88%
rubixi	338.3390	49.2120	223.6430	8.0032	- 33.90%
suicide*	3.3814	0.0849	3.8928	0.0906	+ 1.60%
timelock	8.1895	0.1480	8.1075	0.1149	- 1.00%

Table 8.1: Execution times for Storage and Transient Storage

Table 8.1 shows the obtained mean execution times as well as the standard deviation for the 10 runs of the tool on both versions of the contract. The table list all contracts published as Solidity examples alongside Mythril. The contracts in the list are named after the solidity files. Some contracts are indexed, indicating that a specific solidity file contains multiple contracts and hence results in multiple bytecodes when compiled. An asterisk (*) next to the contracts name means that this specific contract does not use SSTORE, SSLOAD, TSTORE or TLOAD opcodes, meaning the difference in execution time is not the result of the modeling of either storage or transient storage. As shown in the table, for the contracts marked with an asterisk, the percentage differences in execution time are very low (less than 2%).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Comparing the execution times in the table, we see that for the majority of contracts, the execution on the contract's version using transient storage completed faster. This is visible in the last column, showing the difference in percent between the storage mean execution time and the transient storage mean execution time. Negative values refer to a drop of the execution time from the storage version to the transient storage version, meaning the execution of the transient storage version completed faster.

We can identify 2 contracts where the execution time for the transient storage version is higher (indicated by the '+' in the Difference column). However, these two contracts do not contain any storage or transient storage opcodes (indicated by the asterisk near their name). As the absolute execution time is quite low (less than 4 seconds mean execution time on both versions for both contracts) and the increase in percent is also neglectable (+0.09% and +1.60%), the increase in execution time from the storage version to the transient storage version is likely the result of noise.

Overall, we can see that there are multiple contracts with low mean execution times (less than 10 seconds). Our implementation of transient storage modeling seems to have less of an effect (also percent-wise) on the execution times of these contracts. These contracts in the Mythril repository are generally simpler and contain less code (hence the lower execution times).

There are also contracts that have higher absolute mean execution times (more than 10 seconds, with some more than 100 seconds and 2 of those with more than 300 seconds). Inspecting these contracts in the Mythril repo shows that these are contracts with a lot more code and more interactions with storage, using Solidity mappings and structs. The percentage differences of the execution times for these contracts are significantly higher, with execution time reductions of more than 30% for some of the contracts and up to 57% for one of the contracts.

8.5 Hardware Setup

The evaluation of the tool utilized a system with an AMD Ryzen 5 PRO 4650U with Radeon Graphics CPU, featuring a base frequency of 1.4 GHz and a maximum boost of 2.1 GHz across 6 cores. The system's memory includes 16 GB of DDR4 RAM at 3200 MT/s. The GPU is an AMD/ATI Renoir. Storage is provided by a 476.9 GB NVMe SSD. The operating system used for testing is Kali GNU/Linux Rolling 2023.4.

8.6 Limitations of the Evaluation

This section discusses some limitations associated with the evaluation process. These limitations are a consequence of deliberate methodological and practical considerations, essential for the scope and feasibility of this thesis. Acknowledging these constraints is vital for the interpretation of the evaluation findings and the identification of areas for future research.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

We focus on two primary limitations:

- Even though the modeling of storage can serve as baseline for evaluating the modeling of transient storage — since storage opcodes behave equivalently to transient storage opcodes for the duration of one transaction —, it does not allow to draw a clear conclusion with regards to the comparison in terms of efficiency. This is because even though the opcodes of storage and transient storage behave similarly in the EVM, the modeling for storage in state-of-the-art symbolic execution tools oftentimes includes additional features. One example of such a feature is the dynamic loading of on-chain values (obviously not relevant for a transient, per-transaction data storage). These features increase the complexity of the modeling and can therefore reduce its efficiency.
- At the time of writing this chapter, transient storage has been incorporated into Ethereum, but it is still quite new and the available benchmark solidity examples on Mythril's Github do not yet contain transient storage use cases. As the topic of this thesis is not the creation of new benchmark smart contracts, we chose to use the smart contracts that are currently available on the Mythril Github repository.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 9

Discussion

In the following section, we discuss the implications of our findings in terms of the research questions we defined. We also discuss potential future research with regards to transient storage in symbolic execution tools.

9.1 Implications of Findings

- *RQ1: Which modeling techniques do state-of-the-art security analysis tools use to navigate the trade-off between accuracy and computational efficiency during symbolic execution to effectively identify vulnerabilities in smart contracts?*

  For security analysis tools, accuracy is essential. Inaccuracies resulting from imprecise modeling of data storage areas are problematic, as data storage areas are relevant for security (e.g. one of the intended use cases of transient storage are reentrancy locks). This is also reflected in the care that is taken by state-of-the-art tools: Our comparative analysis in chapter 4 has shown that symbolic execution tools mostly rely on SMT arrays (a data structure capable of handling SMT constraints accurately) and only deviate from this option to increase efficiency if the probability of inaccuracies is low (Mythril uses a dictionary instead of an SMT array for memory). Therefore, our methodology advocates for a judicious application of abstractions to maintain a balance between accuracy and computational efficiency. This careful approach ensures that abstractions are employed only under circumstances where inaccuracies are assessed to be unlikely, thereby upholding accuracy as a priority while improving efficiency.

- *RQ2: What factors need to be considered when modeling transient storage in a symbolic execution tool to detect vulnerabilities both accurately and computationally efficiently?*

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek, WIEN Your knowledge hub

- The functionality that has been defined for transient storage and its opcodes in the EIP [AS18]. As the modeling simulates what happens in the EVM, the technical specifications of transient storage should be taken into consideration. This includes aspects like the fact that transient storage maps 256 bit addresses to 256 bit values, the interaction of TSTORE/TLOAD with the stack, and the gas costs of these operations.
- The use cases that transient storage is intended for. As our comparative analysis of state-of-the-art tools has shown, the decision whether to use an SMT array versus a dictionary depends specifically on whether the use cases lead to different but equivalent symbolic indices, which in turn depends on whether the data in the respective data storage type is persisted across transactions. The fact that transient storage has a rather short-term purpose and is not persisted across transactions should be taken into consideration.
- The specific symbolic execution tool that transient storage is being integrated in. The different state-of-the-art symbolic execution security analysis tools for smart contracts pursue the same overall goal, but each uses its own symbolic execution engine, is structured differently and features different functionalities. This should impact the decision making of how to model transient storage, as a data storage area needs to be integrated in the existing architecture of the tool, needs to work in lockstep with the symbolic execution engine and needs to be tailored to the (sometimes unique) features of the overall tool.

• *RQ3: What implications do the identified factors have for the implementation of the transient storage modeling?*

  The technical specifications (size of addresses and storage slots, interactions with the stack, gas costs, etc.) should be modeled accurately, as not doing so will simulate the execution of a transaction improperly, thereby leading to inaccuracies in the identified vulnerabilities.
  Furthermore, the fact that transient storage is not persisted across transactions enables the option of utilizing a dictionary for modeling transient storage.

• *RQ4: Which of the identified modeling techniques is the most appropriate for the implementation of the modeling of transient storage in a symbolic execution tool?*

  The use of a dictionary instead of an SMT array allows for gains in efficiency without lowering accuracy. The evaluation has shown that for larger contracts with longer execution times, the efficiency gains were more substantial than for smaller contracts with shorter execution time.

9.2 Future Research

As transient storage has been incorporated into Ethereum only very recently (March 2024) and only little concrete implementations (and no benchmark smart contracts) of

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

transient storage use cases exist publicly (especially of the more complex ones), it will be interesting to evaluate how significant the efficiency gains of our implementation of transient storage modeling are in absolute terms on contracts that represent the complex use cases of transient storage.

Another potential topic of future research also depends on the integration of transient storage into Ethereum: As soon as various implementations of transient storage use cases emerge, it will be interesting to evaluate if using a dictionary instead of an SMT array for the modeling of transient storage still behaves accurately and vulnerabilities are identified correctly. This will to a large degree depend on whether different, but potentially equivalent symbolic indices used on transient storage will emerge during symbolic execution of those specific implementations of the use cases.

A third potential topic of future research results from the limitations of some symbolic execution security analysis tools, such as Mythril, which do not support the analysis of transient storage use cases involving multiple contracts due to its inability to process separate contracts in a single analysis. Transient storage use cases however are relevant specifically when separate contracts interact with each other. Even though this does not make Mythril's analysis irrelevant (as analyses on single contracts still yield meaningful results), the symbolic execution might have to deal with different obstacles in a tool that does support this feature. In a tool that can process separate contracts in a single analysis, the problem of different, but potentially equivalent symbolic indices might be more likely to arise due to the increased complexity. This would potentially rule out the option of using a dictionary to model transient storage in such a tool.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Figures

6.1 Data Storage in Mythril 24
6.2 Adjusted EVM With Transient Storage [Pas] 25

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Tables

8.1 Execution times for Storage and Transient Storage 37

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Smart Contracts

7.1 Simple Case 29
7.2 No Persistence across Multiple Transactions 30
7.3 Potentially Equivalent Symbolic Indices 31
7.4 HashesAsIndices 32
7.5 Arbitrary Write to Transient Storage 32
7.6 Access to Transient Storage after External Call 33

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Code and Data Availability

The code and the evaluation data is available publicly at `https://github.com/JaakWeyrich/mythril-transient-storage`.

Structure of the Repository

The repository is structured similarly to the original Mythril repository. The python code from the tool itself is inside the `mythril` directory. The changes made to this python code in the context of this thesis are described in chapter 6.

There are additional folders that have been added to the repository in the context of this thesis, specifically `evaluation` (which contains the evaluation data) and `bytecodes` (which contains the bytecodes necessary to carry out the evaluation).

Structure of the Evaluation Directory

The evaluation directory contains a script that was used to carry out the evaluation semi-automatically (it has to be adapted and run manually per evaluated contract).

The evaluation directory additionally contains all the output (structured by evaluted contracts) from the script mentioned above, specifically the identified vulnerabilities as well as the data indicating the computational efficiency.

The evaluation directory also contains one auxiliary script for extracting all the relevant data on computational efficiency to aggregate it in one file for better overview.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Overview of Tools Used

Generative AI Tools

Here is a list of tools that were used in the creation of this thesis:

- ChatGPT-4 and ChatGPT-4o

The listed AI tools were used as supportive aids for several purposes, in particular:

- For the initial analysis of code (specifically during the analysis of the existing symbolic execution tools) and to obtain a basic first overview of code.
- For research purposes, with the information obtained always being verified through additional sources.
- For the creation of auxiliary scripts for the evaluation (e.g. the conversion of storage bytecode into transient storage bytecode).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliography

[ang16] Angr: A Next-Generation Binary Analysis Platform. https://github.com/angr/angr, 2016. Accessed: 2024-02-26.

[AS18] Alexey Akhunov and Moody Salem. EIP-1153: Transient storage opcodes [DRAFT]. `https://eips.ethereum.org/EIPS/eip-1153`, 06 2018. Ethereum Improvement Proposals, no. 1153.

[BCD+11] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, pages 171–177. Springer, 2011.

[But14] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. White Paper, 2014. Available at: https://ethereum.org/en/whitepaper/.

[CARB12] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394. IEEE, 2012.

[Cha23] ChainSec. Documented timeline of defi exploits. https://chainsec.io/defi-hacks/, 2023. Accessed: 2024-01-26.

[Con18] ConsenSys. Mythril: Security analysis tool for evm bytecode. https://github.com/Consensys/mythril, 2018. Accessed: 2024-01-26.

[DAS19] Monika Di Angelo and Gernot Salzer. A survey of tools for analyzing ethereum smart contracts. In *2019 IEEE international conference on decentralized applications and infrastructures (DAPPCON)*, pages 69–78. IEEE, 2019.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[Flo67] Robert W Floyd. Assigning meanings to programs. In *Program Verification: Fundamental Issues in Computer Science*, pages 65–81. Springer, 1967.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

[God11] Patrice Godefroid. Higher-order test generation. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 258–269, 2011.

[Kin76] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[KR18] Johannes Krupp and Christian Rossow. {teEther}: Gnawing at ethereum to automatically exploit smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1317–1333, 2018.

[man19] Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. https://github.com/trailofbits/manticore, 2019. Accessed: 2024-02-26.

[MMH+19] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189. IEEE, 2019.

[NPWB18] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. Btor2, btormc and boolector 3.0. In *International Conference on Computer Aided Verification*, pages 587–595. Springer, 2018.

[Pas] Pascal Marco Caversaccio. Adjusted EVM with transient storage - Fellowship of Ethereum Magicians forum post. Fellowship of Ethereum Magicians forum. Accessed: 2024-01-26.

[PDT+20] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE symposium on security and privacy (SP)*, pages 1661–1677. IEEE, 2020.

[Rem] Remix Project. Remix integrated development environment. `https://remix.ethereum.org`. Accessed: 2024-02-26.

[SWS+16] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE symposium on security and privacy (SP)*, pages 138–157. IEEE, 2016.

[Tan19] Wei Tang. EIP-2200: Structured Definitions for Net Gas Metering. `https://eips.ethereum.org/EIPS/eip-2200`, 07 2019. Ethereum Improvement Proposals, no. 2200.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[tee18] teether: Gnawing at Ethereum to Automatically Exploit Smart Contracts. https://github.com/nescio007/teether, 2018. Accessed: 2024-02-26.

[Wey23] Jaak Weyrich. mythril-transient-storage: Enhancements for mythril classic to incorporate transient storage analysis. https://github.com/JaakWeyrich/mythril-transient-storage, 2023. Accessed: 2024-02-26.

[Woo23] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. https://ethereum.github.io/yellowpaper/paper.pdf, 2023. London Version d01f0fd.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub