Sound analysis and extraction of data from Smart Contract

Maha Ayub
maha.ayub@itu.edu.pk

Information Technology University

Waiz Khan
Information Technology University

Muhammad Umar Janjua
Information Technology University

Research Article

Keywords: Blockchain, Smart Contract, State Extraction, Migration, State verification

Posted Date: August 14th, 2023

DOI: https://doi.org/10.21203/rs.3.rs-3241488/v1

Additional Declarations: No competing interests reported.

Version of Record: A version of this preprint was published at Automated Software Engineering on February 29th, 2024. See the published version at https://doi.org/10.1007/s10515-024-00422-3.

Sound analysis and extraction of data from Smart Contract

Maha Ayub[1*], Waiz Khan[1†] and Muhammmad Umar Janjua[1†]

[1*]Department of Computer Science, Information Technology University, Lahore, Punjab, Pakistan.

*Corresponding author(s). E-mail(s): maha.ayub@itu.edu.pk;
Contributing authors: waiz.khan@itu.edu.pk; umar.janjua@itu.edu.pk;
[†]These authors contributed equally to this work.

Abstract

With the addition of multiple blockchain platforms in the ecosystem, the Dapp owners need to migrate their smart contracts from one platform to another to remain competitive, cost-effective, and secure. A smart contract is a piece of code that contain logic and data. To migrate a smart contract, whether it's on the same blockchain platform or a different one, we need both its source code that represents the logic and data which indicate the state of the contract. The source code can be easily set up, but to complete the migration, we have to extract the current state of the contract. In this paper, we have developed an advanced state extraction technique that uses static analysis to analyze the smart contract's call graph and events, and extracts the entire storage state from the storage trie, along with the proper associations across function calls, enabling users to visualize, manage, and transform the state as desired for migration. The soundness of the extracted state was confirmed using the method of abstract interpretation. Further, the migration adapter is designed that transform extracted state into slot-value pair and migrated it to the target blockchain. Our new approach has allowed us to analyze 14% more smart contracts with the extraction of 15% more data from 67,993 contracts, and migrate some of them to the Polygon test-net.

Keywords: Blockchain, Smart Contract, State Extraction, Migration, State verification

1 Introduction

Ethereum launched in 2015, is one of the largest Blockchain platforms in the market [1]. With Ethereum, developers gain the facility to design decentralized applications using

smart contracts. Smart contracts revolutionized many business applications by removing the need for third parties, e.g., in payment applications, insurance claims, crowdfunding, and supply chain In addition to Ethereum, there are multiple blockchain platforms that support and facilitate the smart contract execution. To remain competitive, secure, and cost-efficient, the smart contracts that rely on them will be needed to migrate from costly blockchain platforms to other ones. However, the differences in data representations, consistency, and immutability introduce unique challenges when migrating smart contracts [2][3]. Smart contract migration not only involves the redeployment of source code on other blockchain networks but also needs a mechanism for the migration of state along with source code. Mainly, smart contracts use a key-value structure to store and maintain the set of states in their embedded storage trie, and migrating smart contracts means reestablishing their state to the targeted blockchain.
Current migration practices rely on wallets [4][5][6] and bridges [7][8], which only give the facility of atomic cross-chain swaps of assets and tokens. Along with assets and tokens, smart contracts' other state/data that involves complex mapping structures must also be transferred for a complete migration, which still depends on manual approaches [9]. Some platforms like Polkadot and Cosmos provide the facility of passing state information from one blockchain to another, but this facility is applicable within their platform. Although current technologies help pass information and synchronize data, they are not suited for transforming and transferring large amounts of information at once. Moreover, due to platform changes, data inconsistency in the targeted blockchain also needs to be handled. Due to these migration challenges, there is a need to design a technique that automatically extracts all smart contract states with the proper association that can be easily transformed and migrated in a single shot. This observation is supported by recent work [10], which proposed an automatic tool to extract smart contracts with their associated variables. However, their extraction algorithm is limited to some extent. Furthermore, they only mention migration as a use case but do not provide any automatic mechanism.
Limitation of previous study: The state of a smart contract represent by its state variables and their extraction involves getting values from their respective slots in storage-trie. The values of elementary state variables are easily extracted from their single assigned slot, but for mapping variables, we need the list of mapping keys to get the desired slots and their respective values. The previous proposed key approximation algorithm uses the control flow graph to backtrack the origin of mapping keys, then keys are further extracted from the transactions. However, the key approximation algorithm that determines the mapping key's origin of the mapping state variable is limited to the intra-procedural level, which limits the analysis and skips those keys that comes from the function calls. Additionally, due to message calls between smart contracts, some keys are still not extractable from transactions, so the emitted events must be analyzed in smart contracts to find the missing keys. In this paper, we provide an inter-procedural state extraction technique that analyzes and extracts the smart contract state. Based on the static analysis, the smart contract source code is analyzed using inter-procedural key approximation, aliasing, and event analysis to determine where mapping keys originate. Further, the mapping keys are extracted from transactions and logs using the extraction algorithm. Then, the extracted state is transformed and ready to migrate into any other EVM platform. For the evaluation of the migration scenario, we also performed a case study on the automatic migration

of smart contracts from Ethereum to the polygon blockchain test-net platform. Migrated smart contracts are also verified through mutations in the state.
Our contribution The main contributions of this paper are:

- Propose an inter-procedural key approximation algorithm including alias and event analysis, which analyze the smart contract source code and provide the origin of mapping keys for each mapping variable.
- Provides an extraction mechanism that analyzes the transaction and logs of a smart contract and extracts the mapping keys along with their corresponding values.
- Provides the migration adapter that migrates the smart contract state from Ethereum to other blockchain platforms in a single transaction.
- We also provide the safety and soundness guarantees of smart contract mapping key extraction using Abstract interpretation.
- The Mapping key extraction algorithm was experimentally evaluated by successfully analyzing 67,993 smart contracts with mapping data structures. This resulted in the extraction of 15% more keys and a 14% increase in the number of fully analyzed smart contracts compared to the previous approach.
- We also performed the case study by selecting some smart contracts from our dataset on the basis of different data sizes and successfully migrating from Ethereum to Polygon test-net.

The rest of the paper is organized as follows. In Section II, we outline the reasons for migrating from Ethereum to other EVM-compatible blockchains. and describes the limitations of previous migration approaches. Section 3 describe and discusses the workflow, design, and the major algorithms of state migration. We present the algorithms evaluation results along with a case study of smart contract state migration from Ethereum to Polygon in Section 5. Lastly, we conclude our paper in Section 6.

2 Motivation & Related work

Similar to how regular databases are moved from local servers to cloud services or from one cloud provider to another to make them more efficient, improve performance, or reduce expenses, the blockchain ecosystem also offers different platforms for users to migrate their smart contracts or decentralized applications from one platform to another. This migration allows users to choose platforms that align with their preferences for performance or cost. Ethereum is currently the most widely-used platform for deploying decentralized applications, but its popularity and increasing demand can impact both its cost and performance. For example, the activation of Ethereum's London upgrade [11] at the beginning of August makes the transaction fees more predictable. Also, the increased demand for block space kept the transaction fees higher than before the hard fork. Many users are unable to bear the cost of sending transactions due to transaction fees. Moreover, High demand also leads to slower transaction execution. In response, a significant number of assets and users migrated to alternative blockchains that are compatible with Ethereum smart contracts [12].
Smart contracts have a higher chance of migration to EVM-compatible blockchains because the entry barrier for existing smart contracts is low since the same code works

on both platforms. Second, the higher availability of Ethereum development tools and services on the other EVM-compatible blockchains. Moreover, no additional expensive and time-consuming audits are required, allowing existing smart contracts to be deployed swiftly[13]. The major EVM-compatible blockchains that users consider based on services and facilities are Avalanche Contract Chain (C-Chain), Binance Smart Chain (BSC), and Polygon. As shown in figure 1 that the transaction cost during the given period on the Ethereum blockchain is extremely high compared to other blockchains, whereas the polygon is the lowest also with the highest execution rate. This motivates us to offer users a reliable method of migrating smart contracts across EVM-compatible blockchains. However, the migration of smart contracts from Ethereum is not an easy process due to

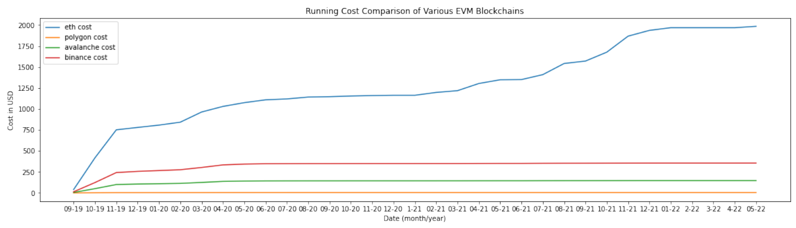


Fig. 1: Transaction cost of different EVM-compatible blockchains during the time period (2019-2022).

immutability, consistency, and data representation barriers. EVM compatibility makes it easy to deploy smart contract source code as it is or with changes. However, the availability of state/data of the smart contract with the proper association needs to be handled for migration according to platform requirements. Smart contract assets/tokens migration can be done easily using multiple bridges and wallets. Mostly these bridges are based on the atomic swap [14] that rely on the specific standard function like ERC20 to make it applicable, if the user does not follow the standard then it may not be able to transfer their assets.

The state migration does not only include the transfer of assets but it is also required to migrate all other state data/information of the smart contract. Some multi-blockchain frameworks like PolkaDot [15] provide the facility of data transfer using a central relay blockchain called parachains. The relay blockchain enables interchain communication of parachains by a message-passing protocol. Also, the parachains have to comply with specific interfaces to interact with the relay blockchain. However, existing blockchains like Ethereum will have to be integrated via bridge blockchains, which only provide a specific token migration facility. Cosmos [16] is another project aiming to transfer information between blockchains. Similarly to PolkaDot, Cosmos also enables interchain communication. Moreover, Cosmos also does not enable data migration or communication between existing blockchains out of their ecosystem. Another approach [17] is presented that syncs or transfers the data between blockchains, but their state extraction technique requires replaying all transactions, which increases the execution and storage overhead if transactions are in million. Moreover, they rebuilt the state on the target blockchain by applying all state transitions sequentially, which directly impact the migration costs. All the bridges or platform focus on cross-chain communication that helps in transferring the data but does not provide an automatic and verified mechanism of data/state extraction from the smart contract. As migration is one shot process, we need a technique that extracts the complete data from the smart contract storage trie and migrates it in one shot with minimal manual involvement.

3 Methodology

For migrating the state to other EVM-compatible platforms, we need to analyze, extract, and pack the smart contract storage state and then deploy it on a targeted blockchain. It is an automatic process that analyzes the smart contract's source code and retrieves real values from its storage trie by getting the slot number and list of mapping keys for mapping variables. Further, the extracted state transforms into slot-value pairs and deploys on the target Blockchain to complete the migration procedure, as shown in Figure 2.

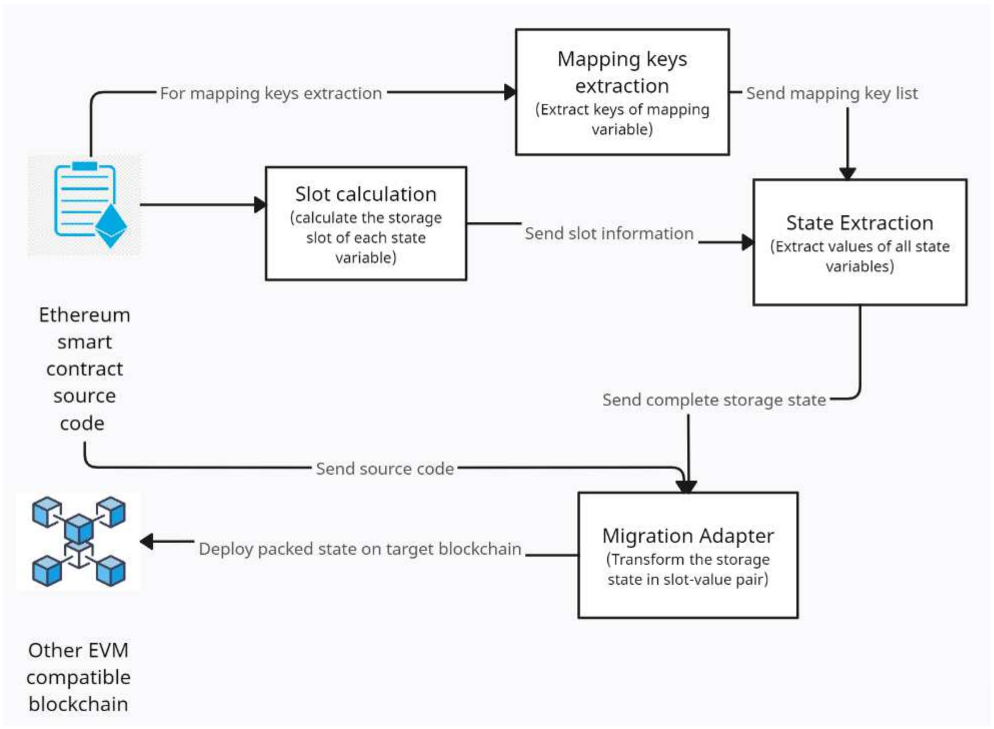

Mapping keys extraction
(Extract keys of mapping variable)
For mapping keys extraction
Send mapping key list
Slot calculation
(calculate the storage slot of each state variable)
Send slot information
State Extraction
(Extract values of all state variables)
Ethereum smart contract source code
Send complete storage state
Send source code
Migration Adapter
(Transform the storage state in slot-value pair)
Deploy packed state on target blockchain
Other EVM compatible blockchain

Fig. 2: State Migration Flow.

The Migration process is divided into four sub-processes as described below:

- **Slot calculation:** Provides the slot number of all state variables by analyzing smart contract Abstract syntax tree (AST).

- **Mapping key extraction:** Analyzes the sources of mapping variables' keys using inter-procedural, event, and alias algorithms for extracting mapping keys.
- **State Extraction:** It provides the values of all state variables from the smart contract storage-trie using slot calculation and mapping key extraction information.
- **Migration Adapter:** Transforms extracted data into a slot-value pair and migrates it to the target blockchain.

The following subsections give detailed technical aspects of the above-mentioned processes.

3.1 Slot calculation

A smart contract's assigned storage trie after its deployment holds the permanent space for each state variable. In a storage trie, each slot can hold 32 bytes, and a trie can be as large as 2^{256} [18]. State variables in the smart contract are assigned slots strictly in the order in which they are declared. A variable's type and size determine the number of slots. Using the Slot calculation algorithm [10], we get the list of all slot numbers of state variables. The algorithm iterates over the Abstract Syntax Tree (AST), identifies all state variable declarations, and provides the list of slot number of state variables on the basis of their declarations.

3.2 Mapping key Extraction Algorithm

Unlike non-mapping variables, mapping variables is a complex data structure that stores their values on hashes, which are the combination of assigned slot number at compile time and inserted mapping keys during the runtime [19]. To find the mapping variable slots, we need all keys that are inserted in the mapping variable during the lifetime of the smart contract. For extracting the mapping keys, the inter-procedural key approximation analyzes the source code and approximates the origins of all mapping keys, which can be further extracted from the respective source. As shown in Figure 5, the key approximation analysis uses the CG (call graph) and CFG (control flow graph) to determine the origin of each mapping variable across the function call. The event and alias analysis are further applied to analyze the storage references of mapping state variables and events. Afterward, a list of approximated mapping keys is extracted from their respective origins including blockchain transactions and logs results.

3.2.1 Inter-procedural key Approximation Analysis

The mapping data structure does not store its keys in the storage trie. It is crucial to identify the correct and precise mapping keys in order to recover the mapping variable. To approximate the keys for mapping variables, we devise a static and source code-based *Inter-procedural key approximation* algorithm. This algorithm determines whether a mapping key was declared as a known static constant value (C) or an unknown value that cannot be extracted at the static time (τ). The know constant will either be any state variable value, comes from a function argument, or a hard-coded value in the source code. There are some pre-processing steps beforehand:

1. Mark the functions that change the mapping variables.

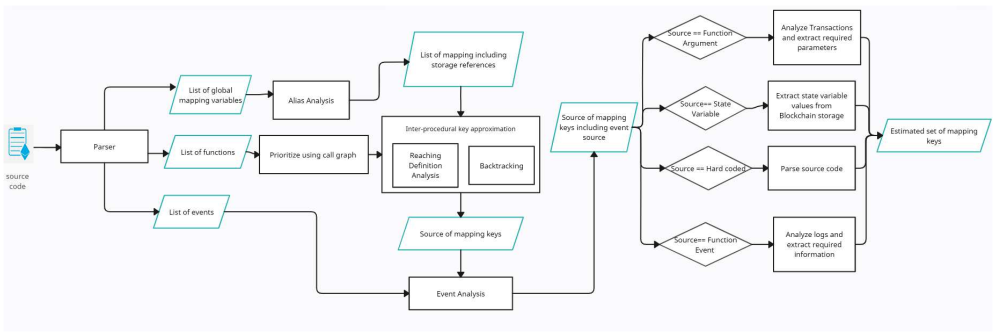

source code
Parser
List of global mapping variables
List of functions
List of events
Alias Analysis
Prioritize using call graph
List of mapping including storage references
Inter-procedural key approximation
Reaching Definition Analysis
Backtracking
Source of mapping keys
Event Analysis
Source of mapping keys including event source
Source == Function Argument
Source== State Variable
Source == Hard coded
Source== Function Event
Analyze Transactions and extract required parameters
Extract state variable values from Blockchain storage
Parse source code
Analyze logs and extract required information
Estimated set of mapping keys

Fig. 3: Mapping key extraction flow.

2. Generate the call graph of the function and prioritize the list of functions on the depth-first approach in which the leaf node functions come first in the list. During the analysis, if the caller function is not in the list of marked functions then it is added on the basis of its priority.
3. Generate CFG for each function of the smart contract.

After preprocessing steps, Reaching definition analysis is applied to the function which provides all possible reached definitions of a mapping key that insert in the mapping variable. Further, these definitions trace back to their source, while if it depends on the function call, the backtracking algorithm replaced the value of the definition with the results of the function (which was already analyzed due to prioritization). The prioritization gives us benefit here by providing already analyzed results of the calling function.

```
function f1(uint m,uint n) // Node 0
{ uint p=0; // Node 1
if (balance%2 == 0){
    p=2;}  // Node 2
else if (balance%3 == 0){
    p=3;} // Node 3
else{
    p= f2(m);} // Node 4
M[p]=n; } // Node 5

function f2(uint q) //Node 6
uint i=0; // Node 7
if (block.number%2 == 0){
    i=q;} // Node 8
else{0
    i=5;} // Node 9
return i;} // Node 10
```

Fig. 4: An example of Solidity code where the value of a key is dependent on multiple paths including function call.

Figure 4 illustrates a scenario where the value of the key ‘*p*’ is dependent on different paths of execution including function call, and the value of ‘*p*’ cannot be determined precisely without the help of the proposed inter-procedure key approximation algorithm.

Example 1. *In function 'f1' shown in Figure 4, the values of variable 'p' comes from multiple paths. The table shows the definition set of each variable from all possible paths and their backtracking results on the control flow graph at every node.*

For example, from Node 1 (N_1) there are three possible execution paths each one assigning a different value to variable 'p' hence at Node 5 (N_5) definition set can have 3 different nodes in the definition set. To resolve the function call the reaching definition analysis marks that node and replaces it with the return value of the function which is calculated during return statement backtracking. The return value can depend upon the context/argument(s) of the function, like at Node 4 (N_4), the function call made to the function '$f2$' with 'm' variable passed as argument. After backtracking the return function '$f2$', we know that the function returns the variable 'i' which value comes either from Node 9 definition or from the argument. Therefore, return values of Function '$f2$' will be the value passed as the argument 'm' at Node 5 and constant value '5' from Node 9. Both of these values will be added to the definition sets for variable 'p'.

After the definition sets are calculated the Backtracking algorithm runs from a bottom-up approach to resolve the definition sets to a possible set of values for function '$f1$'. During backtracking, the algorithm resolves the definition of function call definitions with the actual constant values. For example, at Node 5 the value of function call '$f2(m)$' is replaced by the results of its backtraced value 'C_{arg_1}' and the other definition replace with the actual constant 'C_9.

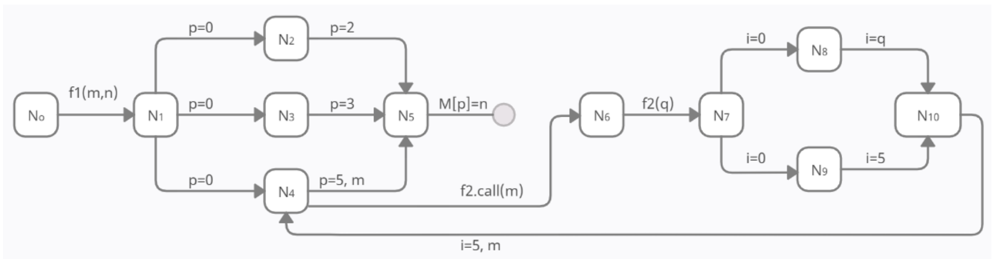


Fig. 5: ICFG(Inter-procedural control flow graph) of the source code mentioned in Figure 4.

Table 1: Table of definition and values at every node of function f1 in the figure 4

Nodes	Definitions	Values
1	$\{m = N_0; n = N_0\}$	$\{a = C_{arg1}; b = C_{arg2}\}$
2,3,4	$\{p = N_1; m = N_0; n = N_0\}$	$\{p = C_0; m = C_{arg1}; n = C_{arg2}\}$
5	$\{p = N_2, N_3, f2(N_9), m; m = N_0; n = N_0\}$	$\{p = C_2, C_3, f2(C_9), C_{arg1}; n = C_{arg2}; m = C_{arg1}\}$

Table 2: Table of definition and values at every node of function f2 in figure 4

Nodes	Definitions	Values
7	$\{q = N_6; i = N_7\}$	$\{1 = C_7; q = C_{arg1}\}$
8,9	$\{q = N_6; i = N_7\}$	$\{i = C_7; q = C_{arg1}\}$
10	$\{q = N_6; i = N_8, N_9\}$	$\{i = C_9, C_{arg1}; q = C_{arg1}\}$

3.2.2 Event Analysis

Messages can be sent by the smart contract to another smart contract or externally owned account (EOA). Unlike transactions, messages are virtual objects that are not recorded in a blockchain during the execution. The recipient's account state will be updated and recorded in the world state if the recipient is EOA. As long as a smart contract receives the message, it is accepted as a function call, and the associated contract code is executed [20]. For swapping or migrating the token from one blockchain to another, communication bridges use the message call between the bridge contract and the token contract [21]. These calls are not registered as transactions of the smart contract, which results in missing some keys during the keys extraction of mapping keys from transactions whose sources are function arguments.
To overcome this scenario, we propose an event analysis algorithm in which we analyze the emitted events in the function. As shown in Figure 6 , the function '_transfer' emit the event of 'Transfer' with the same keys that insert in the mapping variable 'balances'. The algorithm parses the event parameters and back-trace their source in the same manner as we trace back the mapping key sources. If the origin of the event parameter matches the mapping key origin then the event parameters can be extracted from smart contract logs instead of transactions during mapping key extraction process. The Algorithm1 takes the list of marked events that are emitted in the function, and the key source results from the inter-procedural key approximation algorithm as an argument. Next, it extracts the event arguments from an event and backtraces the argument (line 4-5). Further, if these arguments' sources are the same as the key source defined in inter-procedural key approximation results, then they are marked as the event source keys and added to the results (line 6-8). At the key extraction step, if the origin of the key is the event, instead of analyzing the transactions, we analyze the smart contract logs and extract the arguments emitted by the event as shown in Figure 5.

```
contract EventExample {
    mapping (address=>uint) balances;
    event Transfer(address indexed from, address indexed to, uint256 value);

    function private _transfer(address from,address to, uint amount){
        balances[from]=balances[from]-amount;
        balances[from]=balances[to]+amount;
        emit Transfer(from, to, amount);}
        }
```

Fig. 6: Event example.

Algorithm 1 Event Analysis Algorithm

```
procedure EA($markedEvent, keySource$)
Input: Marked event in function, Results of Interprocedural key approximation analysis of a function
Output: Adding event parameter in the key source list.
$event\_arguments \leftarrow parse(markedEvent)$
$earguments\_origin \leftarrow Backtrace(event\_arguments)$
if ($earguments\_origin == keySource$) then
    $keySource.append(event\_arguments)$
end if
return $keySource$
end procedure
```

3.2.3 Alias Analysis for mapping storage references

Local variables can refer to state variables in Solidity, as shown in Figure 7. Variables like these are called storage references. Therefore, a write operation to a local variable is equivalent to a write operation to a state variable. As part of the key approximation algorithm, we compute an alias analysis to find all possible targets of the storage reference. Thus, mapping keys referred to by storage references are also extracted. The Algorithm 2 take the AST of a function and global mapping list as input and provides the updated list of mapping variable that includes all detected storage references of global mappings within the function. First, it extracts the list of local mapping declared in the function in a successive manner by parsing AST (line 4). If the type of local mapping is storage and is initialized with a global mapping variable, then it is a storage reference, and the local mapping variable is added to the list of mapping variables (line 6-8) on which an inter-procedural key approximation algorithm will be performed.

Algorithm 2 Mapping Alias Algorithm

```
procedure CA($AST, ML$)
Input:AST: Abstract syntax tree of the function, ML: list of mapping variables
Output:Updated list of mapping variables including storage references
$local\_mappings \leftarrow parse(AST)$
for all $i \in local\_mappings$ do
    $s \leftarrow i.initialized\_variable\_name$
    if $i.type == 'Storage'$ & $s \in ML$:
        $ML \leftarrow i.name$;
end for
return $ML$
end procedure
```

```
contract Aliasing {
   mapping(uint256 => uint256) public m;
   function StorageReference() public {
    mapping(uint256 => uint256) storage ref=m;
       ref[5]=15;
   }}
```

Fig. 7: Storage reference in the smart contract.

3.2.4 Mapping key extraction

By using inter-procedural key approximation, event analysis, and alias analysis, we obtain the key origin information, which can be further extracted from their respective source. As shown in Figure 5, the keys associated with the function parameter are extracted from the decoded input data of the smart contract transaction, and if the function parameter is also present in the function event, then the value of that parameter is extracted from the smart contract logs also. The keys from the source code can be extracted easily during smart contract parsing. Furthermore, the global variable's origin keys should be extracted from the storage trie of smart contracts. Lastly, we have approximated set of all keys that are inserted into the mapping variable by removing duplicate values.

3.3 Threats to inter-procedural key approximation algorithm

During the inter-procedural key approximation analysis, if the function call is recursive, then the call site is ignored and returns the empty set of definitions. This approach assures us to not get trapped in the loop of function calls but with the cost of making results imprecise. During the evaluation of our approach, we only find the 1 smart contract that contains the recursive call.
The second challenge we face is with loop structures. If the key definitions for mapping depend on the loop index, the algorithm replaces them with τ. In our evaluation dataset, we have identified that 5% of the smart contracts utilize loop structures. To address this, we need to implement a loop analysis technique that can effectively extract the mapping key definitions that are dependent on the loop, which can be done in future work.

3.4 State Extraction

For extracting the state of the smart contract we need the list of mapping keys along with all state variable slot information. Using the information from slot calculation and mapping key extraction information, the state extraction algorithm extracts all values from the storage trie and provided them to the migration adapter for migration to the target blockchain.

3.5 Migration Adapter

To complete the migration process, an adapter is added to the system that deploys the smart contract state on the target blockchain. As defined in Algorithm 3, the migration adapter takes the extracted state, the smart contract source code, migrated function and performs the following task :

- Re-calculate the slots of state variables to accommodate the changes in the extracted state on the basis of user preferences.
- Convert the extracted state into slot and value pair.
- Add a migrated function that contains the assembly command (SSTORE), which inserts the extracted values on the specified slot with once-execute permissions from the owner address.
- After deployment of smart contract, the inserted function is called by the owner of the smart contract that places the extracted data on the storage-trie of smart contract in targeted blockchain

After the successful deployment, the smart contract can be accessible along with all its previous data.

Algorithm 3 Migration Adapter

```
1: procedure MA($state, function, sourceCode$)
2: Input: state: Extracted state of smart contract,
   function: Assembly code that contains SSTORE command,
   sourceCode: Source code of smart contract that migrated to target blockchain.
3: Output: Migrated state to the targeted blockchain.
4: $new\_slots \leftarrow slotcalculation(state)$
5: $slot\_data\_pair \leftarrow conversion(state, new\_slots)$
6: $new\_source\_code \leftarrow sourceCode.added(function)$
7: $new\_source\_code.deployed()$
8: $blockchain.transaction(slot\_data\_pair)$
9: end procedure
```

3.5.1 Security aspects of Migration Adapter

The existing design of the migration adapter assumes that the smart contract owner will act honestly when migrating the state, ensuring secure migration without any alterations. To eliminate owner dependency and implement security measures against state tampering, a thorough security analysis needs to be conducted. However, the primary focus of the current paper is on ensuring the availability of the complete state on the migrated platform and addressing the security aspects of migration can be considered in future research.

4 Abstract Interpretation

In this section, we formalize key approximation analysis and extraction algorithms in previous sections in the context of the Abstract Interpretation Framework [22] to verify the safety and correctness of our extracted state.

4.1 Concrete semantics

To capture the state of the mapping state variable in the source code, we need to track the keys that interact with it. Our solidity language features only have a fixed set of variables that act as a key (it does not feature the structure with multiple fields, values of different sizes, or heap-allocated regions). Therefore, by analyzing all commands '$[\![C]\!]_p$' of function 'm' we have the set of possible reached definitions 'Y' for each mapping key 'X' that is inserted in the mapping variable. These definitions are further traced back to get whether they will be extractable or not and replaced with their origins. The definition '$(k_1, k_2, ...k_n)$' are the constant values that are inserted in the mapping key at each program point 'p' comes from different paths i.e.,

$$[\![C]\!]_p : X \rightarrow Y$$

where

$$Y = \{\{k_1, k_2, ...k_n\}_{p1}, \{k_1, k_2, ...k_n\}_{p2}, ...\{k_1, k_2, ...k_n\}_{pn}\}$$

4.2 Abstract domain

With the sound reaching of all possible definitions at program point 'p' [23], we need to verify whether reached definitions are tracked back to their actual static definitions or not. For mapping the concrete semantics on an abstract domain, we formulate the abstract semantics of the mapping keys which provides the information that the concrete key can be an actual static definition that extracts at the static time or not. The static keys are mapped to the known constant (C_k) which can be further categorized into three basic types as follow:

- Hard-coded (H): These types of definitions are actually hard-coded in the source code of the smart contract. It also includes the local constant and variable values.
- Argument (A): These types of definitions actually originated from the function parameters. They represented the set of definitions instead of a single value and the size of a set is dependent on the number of transactions of a smart contract.
- Global variables (G): These definitions are actually the values of state variables.

The values which cannot be extracted at the static time, we map on the Unknown value (U_k) in the abstract domain. Now, the abstract semantics 'A', which provides the origin of definition 'k' being inserted in the mapping variable at program point 'p' is defined as:

$$A : \gamma(k) \rightarrow AbstractState$$
$$AbstractState \in \{\tau, C_H, C_A, C_G, C_k U_k, \perp\}$$

In Figure 8, the lattice represents the hierarchy of all abstract elements, in which moving down in the lattice represents precise mapping key extraction. The abstract element ($\perp$) in the lattice represents the null key while the key maps on (C_H, C_A, C_G) define the known constant definitions of the mapping key that reached the program point 'p'. The element (U_k) represents that the key cannot be determined at a static time e.g, any runtime variable (like block.timestamp) The top element (τ) is the most imprecise abstract element that represents the set of all possible values that inserts in the mapping key at run-time.

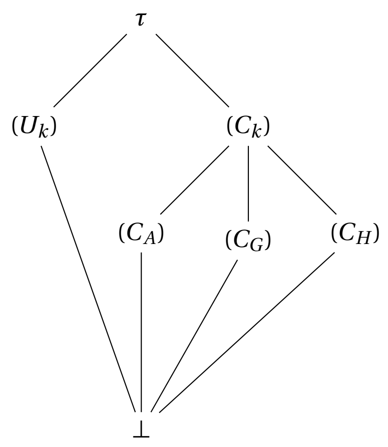


Fig. 8: Abstract domain lattice

4.3 Soundness

Before representing the soundness of static key extraction, we have to define some semantic domains as follows:
$\mathbb{V}$: The set of all state variables
$\mathbb{L}_m$: The set of all local constant values in function ‘m’.
$\mathbb{Y}$: The set of all known constant keys extracted from key static analyzer ($\mathfrak{A}$).
$\mathbb{R}_{env}$: The set of local constant variables whose get value during run-time execution.
$m[S'_v]$: Storage value of state variable v by given slot S'.
$f_c()$: The analyzed function of the smart contract.
$f_c(x_i)$: The input parameter ‘x’ of smart contract function $'f_c'$ at transaction ‘i’.
To make the sound mapping key extraction, we map each concrete element ‘k’ on their most precise abstract element ‘a’ by using α function such as:

$$\alpha(c) = \begin{cases} k = \sim y, \mathfrak{A}(x,y) & \text{a}= \tau \\ k = (x, f_c(x) : x \in \{\bigcup_{i=0}^{n} x_i\}) & \text{a}= C_A \\ k = (m[S'_v] : v \in \mathbb{V}) & \text{a}=C_G \\ k = (l : l \in \mathbb{L}_m) & \text{a}=C_H \\ k = (y, \mathfrak{A}(x,y) : y \in \mathbb{Y}) & \text{a}=C_k \\ k = (input(x) : x \in \mathbb{R}_{env}) & \text{a}=U_k \\ k = null & \text{a}=\perp \end{cases} \tag{1}$$

Using the abstraction function on all mapping keys, we can get abstract set ‘A’ which contains all the possible keys that can be inserted in the mapping variable. Although, some keys in the abstract set may be imprecise, but it guarantees that no key will be missed during extraction. If ‘a’ is the most precise abstraction ($\vDash$) of mapping key ‘k’ then the abstract analyzer holds the property defined as:

$$\forall k \in Y, a \in A, \alpha(k) \leq a \iff a \vDash k$$

For making our abstract domain sound we also need to define the concretization function γ that provides the approximate concrete set of each abstract element:

$$\gamma(k) = \begin{cases} \text{a}=\tau & k = \{\bigcup_{i=0}^{\infty} k_i\} \\ \text{a}= C_A & k = \{\bigcup_{i=0}^{n} f_c(x_i)\} \\ \text{a}= C_G & k = \{\bigcup_{v_i \in \mathbb{V}} m[S'_{v_i}]\} \\ \text{a}= C_H & k = \mathbb{L}_m \\ \text{a}= C_k & k = \mathbb{Y} \\ \text{a}= U_k & k = \mathbb{R}_{env} \\ \text{a}=\perp & k = \{\} \end{cases} \tag{2}$$

The over-approximated key set obtained by concretization holds the actual concrete keys such as:

$$\forall k \in Y, a \in A, k \subseteq \gamma(a)$$

Based on functions (1) and (2) we can establish a Galois connection that would ensure the soundness of static mapping key extraction by including all keys that insert in mapping variable with some false positives.

5 Experimental Validation

In this section, we conduct an experimental evaluation of our techniques using the following research questions:

- *RQ1: How much better is the new algorithm for state extraction 3 in comparison with the previous results [10]?*
- *RQ2: How does the proposed state extraction algorithm perform in terms of time efficiency?*
- *RQ3: What is the overall success rate of the migration process using the proposed migration adapter? How many smart contracts were successfully migrated?*
- *RQ4: What is the cost of migration for the smart contract from Ethereum to the Polygon network?*
- *RQ5: How reliable and accurate are the migrated states on the target Polygon blockchain? Are there any discrepancies or inconsistencies compared to the original states on Ethereum?*

5.1 Experimental Setup

All our experiments were performed on Intel® Core™ i7-8565U CPU at 1.80GHz and 16.0 GB RAM with 512 GB SSD, with Ubuntu 20.04 and Python 3.8.
To extract and migrate state from the Ethereum to polygon blockchain, we used the Python Web3 library [24], and the Infura [25] blockchain API as Web3 providers [26].

Table 3: Comparison of different key stats of Smart contract storage analysis between previous and new proposed algorithms.

Results	**Intraprocedural key approximation analysis**	**Interprocedural key approximation and event analysis**
Total smart contracts analysed	67,993	67,993
Total functions analyzed	1,745,555	1,751,673
Total internal function calls detected	×	653,228
Total event detected	×	207,608
Total mapping references detected	×	0

Table 4: Comparison of extracted mapping keys between previous and new proposed algorithms.

Results	**Intraprocedural based mapping keys extraction**	**Interprocedural & event based mapping keys extraction**
Total mapping keys analyzed	500,527	500,527
Total mapping keys extracted	348,758 (69%)	425,279 (84%) **(Improved by 15%)**
Mapping keys extracted from logs	×	76,521
Mapping keys marked as τ	151,769	75,248
Smart contract marked as τ	18417 (27%)	9202 (13%) (**Improved by 14%**)

5.2 Dataset

We tested our tool on 69,881 unique Ethereum smart contract source codes extracted from the XBlock-ETH dataset [27] which process and provide up-to-date on-chain data from Ethereum. Among the successfully analyzed smart contracts, we also take some smart contracts for migration based on their data size.

5.3 Mapping key extraction evaluation (Response to RQ1)

To measure the impact of our proposed inter-procedural key approximation algorithm and transaction log analysis mechanism, we ran our tool along with the previously proposed technique. Table 3 and 4 show the comparison of the previous technique (intra-procedural key approximation) with the newly implemented inter-procedural technique and event logs analysis.
In both scenarios, 67,993 smart contracts were successfully analyzed out of 68,881, and

failure occurred in other instances due to compilation errors and third-party API failure. A detailed comparison of the source code analysis results of both modes is provided in Table 3.

Table 4 shows the comparison of keys extracted in both scenarios, our tool was able to extract additional 153,042 keys using the inter-procedural analysis and event analysis. Furthermore, in terms of smart contracts, 14% more smart contracts are analyzed completely without τ.

As defined in Section 3.2.1, the mapping key approximation provides the origin of mapping keys, which can then be extracted from transactions and logs. Figure 9 shows the percentage of keys extracted based on origin. With the new inter-procedural algorithm, we can provide more keys in argument key sources, and we are also able to increase our key set by 14% based on the analysis of the events.

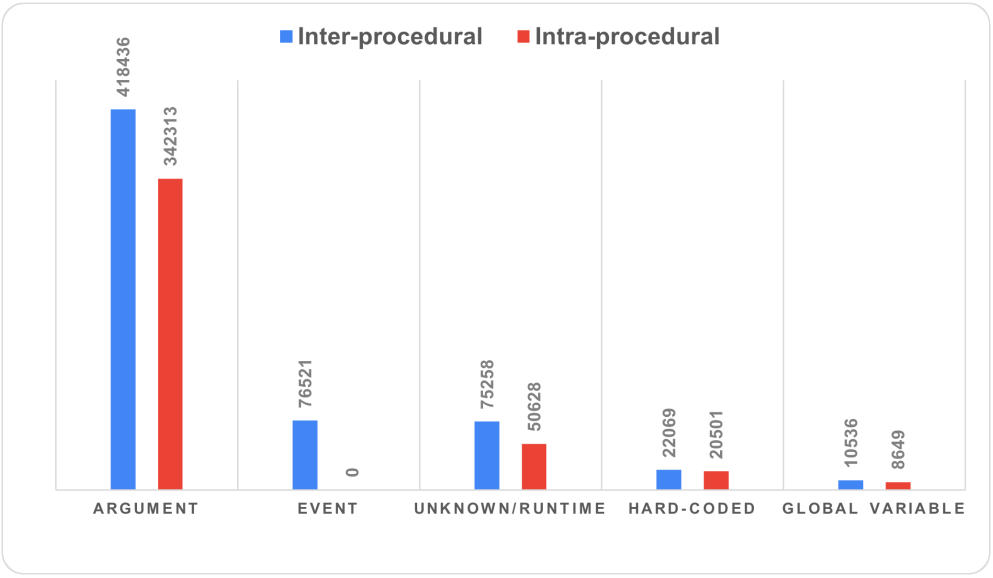
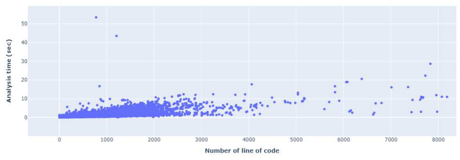


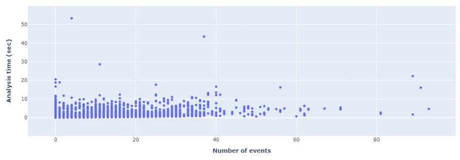
Fig. 9: Distribution of mapping keys on the basis of their source in both previous and new proposed mapping key extraction algorithms.

5.3.1 Analysis time (Response to RQ2):

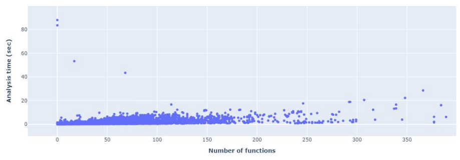
In mapping key extraction, the analysis covered the interprocedural key approximation, event, and Alias algorithm which finds the mapping key source in source code. According to our stats, analysis time does not change with the number of lines of code, functions, events, or state variables, and the analysis results are provided within 20 seconds. Figure 12 represents the analysis time with the lines of code, number of functions, events, and state variables in the smart contract. The y-axis represents time in seconds, and the x-axis represents the count.



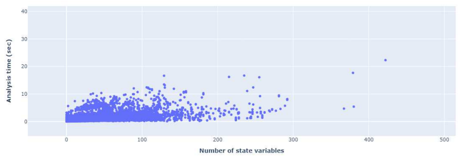
(a) Analysis time vs LOC.



(b) Analysis time vs No. of Events.



(c) Analysis time vs No. of functions



(d) Analysis time vs No. of state variables

Fig. 10: Time performance of proposed key approximations analysis on the basis of an increasing number of events, functions, state variables, and lines of code.

5.3.2 Mapping key extraction time:

The extraction of mapping keys from their sources mainly depends on the transactions and log of smart contracts. For extracting the transaction of smart contracts we use the Infura API that provides 100 transactions per second, increasing the number of transactions directly impacts the mapping key extraction. Our stats show that the smart contract whose transactions are in thousand their extraction time directly increased. Figure 11 represents the mapping key extraction times vs the transactions of smart contracts.

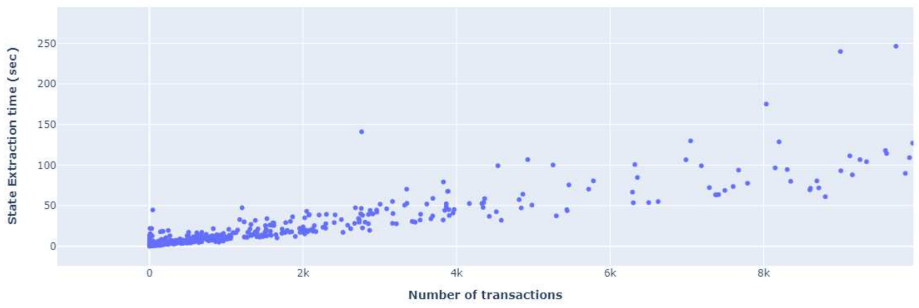
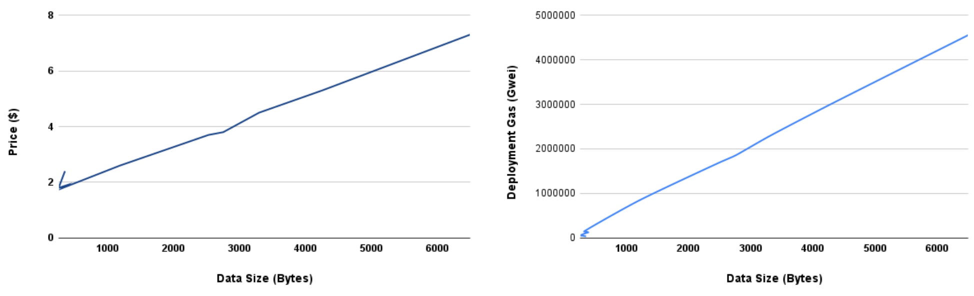


Fig. 11: Performance time of smart contract key extraction of analyzed smart contracts.



(a) Data Size vs Cost. (b) Data Size vs Deplyment Gas.

Fig. 12: The deployment gas and cost of the migrated smart contract on the polygon network on the basis of their size of migrated data.

5.4 State Migration from Ethereum to Polygon (Response to RQ3 and RQ4)

All the smart contracts whose state is extracted can be migrated to the targeted EVM blockchain with its state using a migration adapter. For testing purposes, we choose the polygon network as the targeted blockchain and migrated some selected smart contracts on the basis of different data sizes for estimating the gas cost of migration.
As defined in Section 3.5, we convert the extracted data of the smart contract into slot-data pairs, which consist of all the non-empty slots that represent the state of the smart contract. After the conversion, we added an internal function to the source code that uses the assembly command (SSTORE) to inject extracted data into the desired slot. Our contract migration function takes the smart contract's extracted state, source code, and user's private key, and deploys the source code with the extracted state using the user's address.
The graphs show the deployment gas and price of migrated smart contracts on the polygon network respective to the data sizes of the smart contract. We can see that the increase in the deployment gas depends upon the size of the data injected into the slots of the newly deployed smart contract. Additionally, the increase in size also linearly increases the cost of deployment and deploying a smart contract of approximately 2000 bytes on the polygon blockchain has an average cost of \$3.5[1].

5.5 Validate the migrated state on a targeted blockchain (Response to RQ5)

To ensure the correctness of the migrated state of smart contracts, we establish some criteria to perform some checks:

- The most prominent validation is to make the user access the migrated state with the same identity/address on the targeted blockchain.
- To verify the correct deployment of migrated data, the values of state variables on the target blockchain match the extracted values from the previous blockchain.

[1]The price of migrate state is based on the polygon Matic price on 13 January 2023.

- Ensure the target application works correctly with migrated data.

While mostly EVM blockchain users identify themself with 42 alphanumeric characters that begin with '0x'. As EVM-compatible blockchains, all share the Ethereum address format which gives the liberty to use MetaMask to interact with any other EVM-compatible network using the same address [28]. This gives us the assurance that the migrated data of users is working correctly with the same user address on the target blockchain.
To validate the correct deployment of migrated data, we selected a sample of regular state variables and randomly selected 10 mapping keys of each mapping variable from each migrated smart contract. We then read and compared their values to the extracted values from the previous blockchain. The successful comparison of the selected state variables and mapping key values verify the success of the migration.

The matching of migrated state with the extracted state gives us the assurance of correct state deployment, but we need to ensure that the state is compatible with the target blockchain as well. We solved this problem by making changes or additions to the state and re-extracting it to verify that the state was correctly modified based on the applied changes.
We tested the migrated smart contract 'DecalinxCoin' by first extracting its completed state from the Ethereum main-net, consisting of 79 slots. We migrated the contract with 78 slots after excluding slot of 'owner' variable since we were deploying the contract by our own account to Polygon Mumbai network. We then added new keys to the 'balance' mapping variable through the 'Transfer' function along with modifying some values of mapping keys. At the time of migration migrated data consisted of total 78 slots, and after re-extracting the smart contract from the Polygon blockchain, we found that the state contained a total of 83 slots, including 78 migrated data slots, the 3 slots of additional 3 keys of 'balance' mapping that we added, one slot of owner, and one bool variable named 'contractInitialized' that makes sure contract state is initialized with migrated data only once. We also verified the changes to the 'balance' mapping key values and balance of the owner after calling 'Transfer' function were accurately reflected in the re-extracted state by comparing the new values of the slots.

6 Conclusion

Our proposed technique involves extracting the storage state of a smart contract and migrating it to another EVM-compatible platform. To extract the complex mapping state variables, we use the call graph to identify all mapping keys across function calls. We also perform event and alias analysis to handle message calls between smart contracts and storage references of mapping variables, allowing us to extract the entire state of the smart contract with proper associations to the state variables. Once the extraction is complete, the migration adapter deploys the smart contract on the target blockchain with all the extracted states. In addition, we present an abstract domain for extracting smart contract mapping keys and prove the soundness of our method by establishing a Galois connection between concrete and abstract semantics. Lastly, 67,993 smart contracts were analyzed, and extracted around 15% more data than the previous approach. Moreover, some smart contracts from our dataset were chosen based on size and migrated to polygon testnet at the average cost of \$3.5. To validate the migrated state, we randomly read and matched the values with the extracted state and introduced a mutation, and re-extracted the state from the Polygon ecosystem to determine if our technique could detect the mutation or not. Moreover, as the future direction, our approach not only provides assistance in the secure automatic migration process, but also inspires the concept of "Roamable smart contracts," which can be deployed, run, and switch on any EVM-compatible blockchain based on user preferences. The state of these contracts can be automatically migrated or synced on the current platform.

Declarations

- **Funding:** This research project was partially funded by the National Center of Cyber Security (NCCS) Pakistan.
- **Conflict of interest/Competing interests:** The Authors do not have any conflict to declare.

References

[1] Marr, B.: Blockchain: A very short history of ethereum everyone should read (2018). Accessed: Oct 10, 2020

[2] Bandara, H.D., Xu, X., Weber, I.: Patterns for blockchain data migration. EuroPLoP '20. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3424771.3424796 . https://doi.org/10.1145/3424771.3424796

[3] Paik, H.-Y., Xu, X., Bandara, H.M.N.D., Lee, S.U., Lo, S.K.: Analysis of data management in blockchain-based systems: From architecture to governance. IEEE Access **7**, 186091–186107 (2019) https://doi.org/10.1109/ACCESS.2019.2961404

[4] coinbase: COINBASE wallet (2012). https://www.coinbase.com/wallet

[5] Metamask: The crypto wallet for defi, Web3 Dapps and nfts (2016). https://metamask.io/

[6] Wallet, T.: Best cryptocurrency wallet: Ethereum Wallet: Erc20 wallet (2017). https://trustwallet.com/

[7] Murdock, M.: A deep dive. Avalanche Bridge built by Ava Labs - LI.FI Blog. LI.FI Blog (2022). https://blog.li.fi/avalanche-bridge-a-deep-dive-d51d60dda47f

[8] Binance: Binance: Binance Smart Chain: Binance bridge: Binance Swap: Binance.org. Binance. https://www.bnbchain.org/en/bridge

[9] Josselinfeist: How contract migration works. Accessed: 2020-08-12. https://blog.trailofbits.com/2018/10/29/how-contract-migration-works/

[10] Ayub, M., Saleem, T., Janjua, M.U., Ahmad, T.: Storage state analysis and extraction of ethereum blockchain smart contracts. ACM Trans. Softw. Eng. Methodol. (2022)

https://doi.org/10.1145/3548683

[11] Ethereum: History and Forks of ethereum. Ethereum (2021). https://ethereum.org/en/history/

[12] Polygon: $28b Investment Firm Migrates Equity From Ethereum To Polygon, Upgrading Speed, Liquidity, and Compliance. Polygon Labs (2021). https://bit.ly/44EjdMV

[13] Suisse, B.: Compatible competition - empowering or encroaching on ethereum?: Bitcoin Suisse. Bitcoin Suisse (2021). https://bit.ly/43Au4WQ

[14] Herlihy, M.: Atomic cross-chain swaps. In: Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing. PODC '18, pp. 245–254. Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3212734.3212736 . https://doi.org/10.1145/3212734.3212736

[15] WOOD, G.: POLKADOT: VISION FOR A HETEROGENEOUS MULTI-CHAIN FRAMEWORK (2016). https://polkadot.network/polkadotpaper.pdf./

[16] Kwon, J., Buchman, E.: Internet of blockchains (2021). https://v1.cosmos.network/resources/whitepaper

[17] Westerkamp, M., Küpper, A.: Smartsync: Cross-blockchain smart contract interaction and synchronization. 2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC), 1–9 (2022)

[18] Marx, S.: Understanding Ethereum Smart Contract Storage. "Accessed: Sep 9, 2020" (2018). https://programtheblockchain.com/posts/2018/03/09/understanding-ethereum-smart-contract-storage/

[19] Chriseth, Hari, Mathias, B., Tony: Layout of state variables in storage (2021). https://docs.soliditylang.org/en/v0.8.11/internals/layout_in_storage.html

[20] Makers, D.W.: How Ethereum Manages Transactions. https://coding-bootcamps.com/blog/how-ethereum-manages-transactions.html

[21] Csiro: Token swap (2021). https://research.csiro.au/blockchainpatterns/general-patterns/blockchain-payment-patterns/token-swap/

[22] Cousot, P., Cousot, R.: Abstract interpretation frameworks. Journal of Logic and Computation **2** (1992) https://doi.org/10.1093/logcom/2.4.511

[23] Nielson, F., Nielson, H.R., Hankin, C.: Data flow analysis **2**, 35–139 (19991)

[24] web3py: Introduction. https://web3py.readthedocs.io/

[25] Infura: Ethereum API: Ipfs API amp; gateway: ETH Nodes as a service. https://infura.io/

[26] web3py: Providers. https://web3py.readthedocs.io/en/v5/providers.html

[27] Zheng, P., Zheng, Z., Dai, H.: Xblock-eth: Extracting and exploring blockchain data from ethereum. CoRR **abs/1911.00169** (2019) 1911.00169

[28] Metamask: The Ethereum address format and why it matters when using metamask (2022). https://bit.ly/3Kc40ua