Towards Secure Program Partitioning for Smart Contracts With LLM's In-Context Learning

Ye Liu, Yuqing Niu, Chengyan Ma, Ruidong Han, Wei Ma, Yi Li, Debin Gao, and David Lo, *Fellow, IEEE*

***Abstract*—Smart contracts are highly susceptible to manipulation attacks due to the leakage of sensitive information. Addressing manipulation vulnerabilities is particularly challenging because they stem from inherent data confidentiality issues rather than straightforward implementation bugs. To tackle this by preventing sensitive information leakage, we present PARTITIONGPT, the first LLM-driven approach that combines static analysis with the in-context learning capabilities of large language models (LLMs) to partition smart contracts into critical (privileged) and normal codebases, guided by a few annotated sensitive data variables. We evaluated PARTITIONGPT on 18 annotated smart contracts containing 99 sensitive functions. The results demonstrate that PARTITIONGPT successfully generates *compilable*, and *verified* partitions, achieving a precision of 80% while reducing more than 26% code compared to function-level partitioning approach. Furthermore, we evaluated PARTITIONGPT on nine real-world manipulation attacks that led to a total loss of 25 million dollars, PARTITIONGPT effectively prevents eight cases, highlighting its potential for broad applicability and the necessity for secure program partitioning during smart contract development to diminish manipulation vulnerabilities.**

***Index Terms*—Software engineering, blockchains, smart contracts, trusted computing.**

I. INTRODUCTION

SMART contracts are script programs deployed and executed on blockchain, facilitating the customization and processing of complicated business logic within transactions. Most smart contracts are developed in Turing-complete programming languages such as Solidity [1]. These smart contracts have empowered a wide range of applications across fungible tokens, non-fungible tokens (NFT), decentralized exchanges, and prediction markets on different blockchain platforms including Ethereum [2], BSC [3], and Solana [4]. However, smart contracts may contain design and implementation flaws, making them vulnerable to different types of security attacks. These include common vulnerabilities, such as integer overflow [5] and reentrancy [6], as well as manipulation vulnerabilities, such as front-running [7], user-control randomness [8], and price manipulation [9]. While common vulnerabilities have been extensively studied and well addressed, defending against manipulation vulnerabilities remains a big challenge.

Manipulation vulnerabilities are caused by sensitive information leakage because of the inherent transparency feature of blockchains. For instance, in 2023, Jimbo was attacked due to the manipulation of unprotected price-related internal states called bins, leading to a loss of about eight million dollars [10]. While static analysis [11], [12], [13], [14], [15], [16] and dynamic analysis techniques [17], [18], [19], [20], [21], [22] have demonstrated impressive capabilities in vulnerability detection, they primarily target implementation bugs rather than addressing the inherent risks of data transparency in blockchains. Particularly, Ethainter [14] detects information flow problems in smart contracts, but it is limited to only the detection of unrestricted data write due to poor access control. The price manipulation attacks can be detected by some existing methods [20], [21], [22] through rule-based transaction analysis, but such incomplete rules still leave a room for attackers to adjust their activities to avoid being detected. Similarly, formal verification approaches [23], [24], [25], [26], [27] excel at proving correctness properties but are less effective in scenarios where the root cause lies in unintended exposure of sensitive contract states. Runtime verification techniques [28], [29], [30], [31] offer the potential to halt harmful executions dynamically. However, these methods often require modification of the execution environment or rely on predefined rules that are inadequate for addressing information leakage, as they focus on transaction execution anomalies. Furthermore, program repair solutions [32], [33], [34], [35], [36], [37], [38], [39] concentrate on patching common vulnerabilities or fixing implementation errors. They do not account for design-specific issues related to data transparency, leaving contracts vulnerable to manipulation attacks such as *front-running*, *user-controlled randomness*, and *price manipulation*. Beyond the aforementioned program analyses, a more practical solution is to defend against manipulation vulnerabilities through privacy-aware software development.

The primary limitation of existing approaches to protecting sensitive information in smart contracts lies in their

Received 22 July 2025; revised 28 December 2025; accepted 16 February 2026. Date of publication 2 March 2026; date of current version 20 April 2026. This work was supported by the National Research Foundation, Singapore, and the Cyber Security Agency of Singapore through the National Cybersecurity R&D Programme under Proposal NCR25-DeSCEmT-SMU. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the National Research Foundation, Singapore, and the Cyber Security Agency of Singapore. Recommended for acceptance by H. Zhong. *(Corresponding author: Wei Ma.)*

Ye Liu, Yuqing Niu, Chengyan Ma, Ruidong Han, Wei Ma, Debin Gao, and David Lo are with Singapore Management University, Singapore 188065 (e-mail: yeliu@smu.edu.sg; yuqingniu@smu.edu.sg; chengyanma@smu.edu.sg; rdhan@smu.edu.sg; weima@smu.edu.sg; dbgao@smu.edu.sg; davidlo@smu.edu.sg).

Yi Li is with Nanyang Technological University, Singapore 639798 (e-mail: yi_li@ntu.edu.sg).

Digital Object Identifier 10.1109/TSE.2026.3668858

0098-5589 © 2026 IEEE. All rights reserved, including rights for text and data mining, and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on May 15,2026 at 01:46:10 UTC from IEEE Xplore. Restrictions apply.

coarse-grained handling of sensitive operations. These approaches can be broadly categorized into hardware-supported and software-supported solutions. While the hardware-based solutions [40], [41], [42], [43], [44], [45] provide a trusted execution environment (TEE) that could hide all the execution and data information when running the entire smart contract, they lack flexibility and increase dependency on specific infrastructures. Similarly, the software-supported solutions, e.g., empowered by zero-knowledge proofs [46], [47], seal sensitive operations during execution but often come with computational overhead and integration complexity. These methods fail to leverage the principle of least privilege [48] effectively, as they intermingle sensitive and non-sensitive operations within a single codebase, making it difficult to enforce modular security or validate protection measures in a scalable manner. This intertwining of sensitive and non-sensitive operations complicates security enforcement but also restricts developers from optimally utilizing secure infrastructures. For example, while applications like games can benefit from high-quality randomness by isolating sensitive operations on TEE-based blockchains such as Secret Network [49], manually partitioning the smart contract code is a labor-intensive and error-prone process that introduces potential risks and inefficiencies.

Our work addresses these limitations by introducing an automated and modular framework for smart contract partitioning that is applicable to mitigate manipulation vulnerabilities caused by sensitive information leakage. By combining program slicing and in-context learning capability of large language models (LLMs), we decouple sensitive operations from non-sensitive ones, enabling developers to deploy sensitive operations on secure infrastructures while maintaining non-sensitive operations on standard blockchains like Ethereum. PartitionGPT was designed with two purposes: (1) executing sensitive operations on secure infrastructure prevents the sensitive data to be exposed; (2) proper partitioning minimizes the size of sensitive code and runtime overhead. Firstly, we utilize taint analysis to automatically trace all sensitive operations associated with user-defined secret data in smart contracts, reducing reliance on manual annotations. Secondly, our solution combines program slicing with the capabilities of LLMs to automatically generate fine-grained program partitions that are both compilable and likely secure. This reduces developer effort while ensuring precise partitioning. Additionally, we have developed a dedicated checker to formally verify the functional equivalence between the original and partitioned code, ensuring that the transformed code remains consistent with the intended behavior of the smart contract. This approach effectively addresses the limitations of the existing methods by enhancing security, usability, and reliability.

Our motivation for using LLMs lies in augmenting the automation and flexibility of the partitioning process—especially in smart contract environments where (1) existing program partitioning techniques often assume inter-process communication [50] or trusted runtime enforcement [51], which are not available in public blockchains; (2) smart contracts are written in domain-specific languages, e.g., Solidity, with tight semantic coupling between contract variables, blockchain storage, and transactions, making static code transformation used by existing partitioning techniques brittle [52] or overly conservative [53]. Note LLMs serve only as a search mechanism in the space of candidate transformations, while formal methods enforce correctness.

We implement our approach in PartitionGPT that is able to generate *compilable*, and *verified* program partitions. We evaluate PartitionGPT on 18 annotated confidential smart contracts having 99 sensitive functions in total and nine real-world victim smart contracts of manipulation attacks. The experimental results show that PartitionGPT is able to generate secure program partitions, achieving a precision of 80%, reducing more than 26% code in trusted codebase (TCB) compared to function-level partitioning that does not split sensitive functions. Additionally, PartitionGPT can be applied to detect eight out of nine manipulation attacks, indicating its applicability for protecting real-world smart contracts. We deployed the program partitions within a TEE-based execution environment [41], and it shows the runtime gas overhead for sensitive functions to partition is moderate, resulting in 61% to 103% gas[1] increase for carrying out communications between isolated sensitive code within TEE and normal code on Ethereum. Moreover, our study also compares the performance of four different LLMs. It shows that *GPT-4o mini* used by PartitionGPT outperforms the selected three open-source LLMs where *Qwen2.5:32b* is recommended as the alternative LLM for PartitionGPT.

We summarize the following main contributions.

- To the best that we know, we are the first to propose an LLM-driven framework, PartitionGPT, for secure program partitioning, and apply it to smart contracts. PartitionGPT combines program slicing with LLM's in-context learning for fine-grained partition generation.
- We devise a dedicated equivalence checker for Solidity smart contracts to formally verify the functional equivalence between the original and partitioned code, ensuring their functionality conformance.
- We evaluate PartitionGPT on 18 annotated confidential smart contracts that have 99 sensitive functions. The results show PartitionGPT achieved success rate of 80%, reducing 26% code in TCB. Moreover, our evaluation also indicates that PartitionGPT is able to effectively defend against real-world manipulation attacks, and the runtime overhead of PartitionGPT is moderate by deploying the partitions into a TEE-based secure environment.
- All the benchmarks, source code, and experimental results are available on https://github.com/academic-starter/PartitionGPT.

OrganizationThe rest of the paper is organized as follows. Section II provides the necessary background and justifies the motivation for this work. Section III details our fine-grained program partitioning approach, and we discuss the equivalence checking in Section IV, followed by the implementation and

[1]Gas represents the fee paid for transaction execution on Ethereum or other Ethereum-like blockchain platforms [2].

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on May 15,2026 at 01:46:10 UTC from IEEE Xplore. Restrictions apply.

evaluation in Section V. The related work is discussed in Section VII, and Section VIII concludes the paper and mentions future work.

II. Background and Motivation

A. Blockchain and Smart Contracts, and Their Privacy Issues

Blockchain technology was first introduced in Bitcoin [54] and has emerged as a transformative innovation, enabling decentralized systems that eliminate the need for intermediaries in transactions and applications. At its core, blockchain provides a distributed, immutable ledger that ensures transparency and security. Smart contracts, programmable scripts executed on the blockchain, have further expanded its utility by automating processes such as financial transactions, supply chain management, and governance. These contracts are deterministic and operate transparently, allowing all participants in the network to verify their behavior. This has been pivotal in the success of decentralized finance (DeFi) and other blockchain-based applications, where trust is derived from the open and verifiable nature of smart contracts.

However, privacy is one of the major concerns for blockchains as most of these systems store and log everything viewable to the public [55]. Sensitive information, such as user account details, transaction data, and contract-specific logic, is often exposed on-chain, creating risks of data leakage. For instance, adversaries can analyze public transaction histories and state variables of smart contracts to infer private information in order to exploit vulnerabilities. In DeFi space, attackers have leveraged publicly accessible contract states to orchestrate complex exploits, such as oracle price manipulations and front-running attacks. The lack of mechanisms to distinguish and safeguard sensitive data from non-sensitive data exacerbates these risks, making smart contracts an attractive target for malicious actors.

While efforts to mitigate data leakage risks exist, they are often inadequate. Techniques like homomorphic encryption [56], zero knowledge proofs [46], and multiparty computation [57] can obscure sensitive data, but these approaches may conflict with the principles of decentralization or introduce inefficiencies. This highlights the urgent need for innovative solutions that preserve the decentralized and transparent nature of blockchain systems while providing robust privacy protections. Strategies such as privilege separation could address these challenges, enabling smart contracts to securely handle sensitive information.

B. Manipulation Attacks

Manipulation attacks represent a class of logic-level vulnerabilities in decentralized finance (DeFi) smart contracts, where an adversary strategically interferes with the execution context or state dependencies of the contract to trigger unintended behavior and extract profit. Unlike traditional software bugs, these attacks exploit the interplay between contract logic and external blockchain state, such as token prices, balances, or transaction ordering. For instance, in a price oracle manipulation attack, if an oracle fetches prices from on-chain decentralized exchanges (DEXs), attackers can temporarily inflate or deflate the token price through large trades, influencing the contract's behavior before the manipulated price reverts.

C. In-Context Learning

In-context learning (ICL) is an emerging capability of large language models (LLMs), enabling them to perform new tasks by conditioning on a small number of input–output examples provided at inference time, without requiring gradient updates or fine-tuning. Unlike traditional supervised learning, ICL allows models to generalize to novel domains using only a textual prompt as guidance. This makes it especially appealing for software engineering tasks where labeled data is scarce or highly domain-specific. In the context of smart contract analysis and security, recent work has successfully applied ICL to problems such as invariant generation [58], specification synthesis [25], and vulnerability detection [16].

D. A Motivating Example

Jimbo suffered from a manipulation attack on May 29, 2023, where the attacker gained a profit of 8 million dollars by devising a highly-complicated transaction sequences with crafted inputs [10]. Jimbo is a Self-Market Making Liquidity Bin Tokens (SMMLBTs) [59], where "bin" represents a range of prices, with positions further to the right denoting higher prices. In the Jimbo protocol, rebalancing of asset in the liquidity pool is defined by the different states of the bin including *active bin*, *floor bin* and *trigger bin*. Specifically,

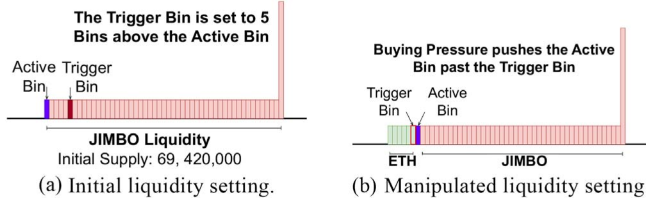
- **floor bin** denotes the minimum price of Jimbo tokens;
- **active bin** represents the price currently traded on; and
- **trigger bin** refers to the price that triggers the rebalancing.

One key point is that when *active bin* is above *trigger bin*, users can call a function *shift()* of Jimbo that is able to increase *floor bin*.

The success of the attack relies on the attacker obtaining exact value of the bins. In this exploit, the attacker first initiated a flashloan of 10,000 Ether to add liquidity to the rightmost bin as shown in Fig. 1(a), where the current the *active bin* is 8,387,711 and *trigger bin* is 8,387,715. Next, the attacker bought Jimbo's token to make *active bin* above *trigger bin*, where active bin moved from 8,387,711 to 8,387,716 as depicted in Fig. 1(b). Subsequently, the attacker rebalanced the liquidity with *shift()* that increased the value of *floor bin*. Hence, the attacker creates a huge arbitrage space, and more details above the following profit earning operations can refer to the security analysis report[2]. Consequently, the attacker sold Jimbo's token at a significantly high price indicated by the increasing *floor bin*. From a software engineering perspective, the vulnerability arises from the tight coupling between price-dependent state variables and privileged update logic, which allows adversarial manipulation to indirectly trigger sensitive execution paths.

[2]https://x.com/yicunhui2/status/1663793958781353985

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on May 15,2026 at 01:46:10 UTC from IEEE Xplore. Restrictions apply.



(a) Initial liquidity setting. (b) Manipulated liquidity setting.

Fig. 1. Illustration of Jimbo's key attack step.

Algorithm 1 Identifying sensitive functions and statements

Require: $\mathcal{V}$, a set of sensitive state variables of smart contract.
$PDGs$, a set of program dependence graphs.
Ensure: $\mathcal{F}$, a set of sensitive functions.
Δ, a set of sensitive statement nodes.

1: $sinkNodes \leftarrow \emptyset$ ▷ Initialize sink nodes
2: **for** $v \in \mathcal{V}, (f, pdg) \in PDGs$ **do**
3: **for** $node \in pdg$ **do**
4: **if** $\exists\ var_rw \in$ node.rwVars, IsDEPENDENT(v, var_rw, pdg) **then**
5: $sinkNodes \leftarrow sinkNodes \cup \{node\}$
6: $\mathcal{F} \leftarrow \mathcal{F} \cup \{f\}$
7: $srcNodes \leftarrow \emptyset$ ▷ Initialize source nodes
8: **for** $sink \in sinkNodes$ **do**
9: **for** $var_rd \in$ sink.readVars, $(f, pdg) \in PDGs$ **do**
10: **for** $node \in pdg$ **do**
11: **if** $\exists var_wrt \in$ node.writeVars, IsDEPENDENT(var_wrt, var_rd, pdg) **then**
12: $srcNodes \leftarrow srcNodes \cup \{node\}$
13: $\mathcal{F} \leftarrow \mathcal{F} \cup \{f\}$
14: $\Delta \leftarrow sinkNodes \cup sourceNodes$ ▷ Include privileged nodes
15: **return** $\mathcal{F}$, Δ

Security Challenges. This delicate attack vector is non-trivial for the existing price manipulation detection tools [20], [22], [21] to detect and the existing runtime verification techniques [29], [31] to defend against. First, the manipulation detection tools heavily rely on the correct extraction of the token exchange rate. But, in this case, the exchange rate between Ether and Jimbo is defined over the complicated states of bin, which is challenging to automatically infer. Second, the manipulation of bin's states is carefully-designed (*active bin* is close to *trigger bin*), making runtime verification tools hard to differentiate between normal transactions and abnormal attacks.

Partitioning-based Solution. Running privileged code inside a secure environment is able to prevent an attacker from reading those sensitive values and therefore the manipulation attack is disallowed. To mitigate such manipulation attacks, in this work, we propose an LLM-driven secure program partitioning method for smart contracts to isolate the execution of the operations related to sensitive data variables, e.g., *floor/active/trigger bin* of Jimbo, within a secure environment, where sensitive information leakage problem could be largely mitigated. We highlight that migrating and running the entire piece of smart contract into the secure environment is not practical, as that will lead to prohibitively high runtime overhead.

III. Approach

A. Overview

Fig. 2 illustrates the workflow of PartitionGPT, an LLM-driven fine-grained program partitioning framework for smart contracts. PartitionGPT processes smart contracts annotated with sensitive data variable information, ultimately producing partitioned contracts. These partitions isolate privileged statements into dedicated subordinate functions, separating them from normal statements. At a high-level overview, PartitionGPT breaks a contract function into two smaller parts–one for normal statements while the other for privileged statements related to operations on sensitive data variables. PartitionGPT encompasses seven steps. ① **Locate** will employ taint analysis to identify critical functions containing privileged statements. For each function, ② **Slice** will yield two program slices according to privileged statements and these slices will be one of prompt parameters for partition generation within ③ **Iterative Loop** for each function. In detail, ④ **Generate** harnesses LLM by incorporating the aforementioned program slices and using a few examples as the seed, thus tailoring the code refactoring process for partition purpose. Syntactically incorrect code alerted by compilers can be revised by LLM with concrete compiler feedback in ⑤ **Revise**. Next, syntactically correct code will be analyzed to determine whether the program is securely partitioned or not in ⑥ **Validate** using an effective detection rule. If it is insecurely partitioned, the current program partition should be repaired. Therefore, we regenerate the program partitions taking the current program partition as bad example. For all the compilable and likely secure partitions, we perform ⑦ **Weighted Selection** to choose the top-K partition candidates. We develop a dedicated prover to conduct equivalence checking between the original and the post-partition function code. Consequently, the correctness of all the resulting program partitions are formally verified and PartitionGPT outputs *compilable*, and *verified* program partitions for smart contracts.

Importantly, if equivalence checking fails, we will discard the partition. This design decision is based on efficiency considerations—equivalence checking is computationally expensive, and our approach relies on the assumption that LLM-generated code is sufficiently diverse. Rather than entering a regeneration loop for each failure, we sample a batch of candidate partitions in advance and use the prover to select only those that pass semantic verification.

B. Illustration Example

We use a case study to illustrate how PartitionGPT generates program partitions. Listing 1 lists the function code of *bid* from an auction contract named *BlindAuction* which is one of official examples provided in Solidity documentation [1]. Briefly speaking, Listing 1 shows that user bids will be processed to update the current `highestBid` and `bidCounter`. When a user have ever put a bid in the auction, `bidCounter` will not be updated. In this auction

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on May 15,2026 at 01:46:10 UTC from IEEE Xplore. Restrictions apply.

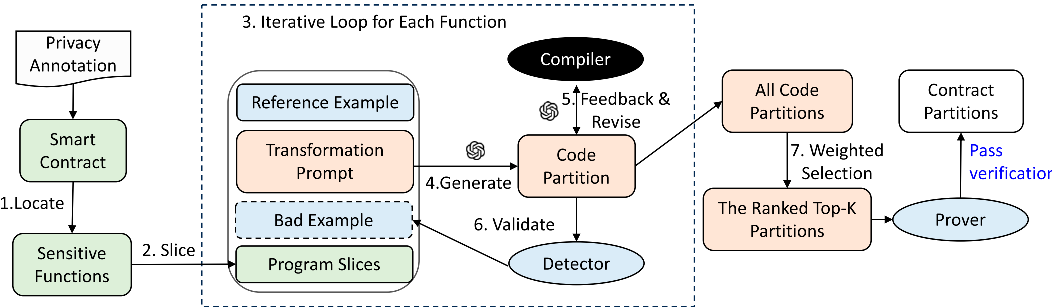


Fig. 2. The overview of PARTITIONGPT.

```
1  function bid(uint64 value) external onlyBeforeEnd
2  {
3   uint64 sentBalance;
4   uint64 existingBid = bids[msg.sender];
5   if (existingBid >0) {
6    uint64 balanceBefore = tokenContract.balanceOf(
         address(this));
7    bool isHigher = existingBid < value;
8    uint64 toTransfer = value - existingBid;
9    uint64 amount = 0;
10   if (isHigher){
11    amount = toTransfer;
12   }
13   tokenContract.transferFrom(msg.sender, address(this)
         , amount);
14
15   uint64 balanceAfter = tokenContract.balanceOf(
         address(this));
16   sentBalance = balanceAfter - balanceBefore;
17   uint64 newBid = existingBid + sentBalance;
18   bids[msg.sender] = newBid;
19  } else {
20   bidCounter++;
21   uint64 balanceBefore = tokenContract.balanceOf(
         address(this));
22   tokenContract.transferFrom(msg.sender, address(this)
         , value);
23   uint64 balanceAfter = tokenContract.balanceOf(
         address(this));
24   sentBalance = balanceAfter - balanceBefore;
25   bids[msg.sender] = sentBalance;
26  }
27  uint64 currentBid = bids[msg.sender];
28  if (highestBid == 0) {
29   highestBid = currentBid;
30  } else {
31   bool isNewWinner = highestBid < currentBid;
32   if (isNewWinner)
33   highestBid = currentBid;
34  }
35 }
```

Listing 1. Original function code *bid* of BlindAuction.

```
1  /* Partitioned Code by PartitionGPT */
2  function bid(uint64 value) external onlyBeforeEnd
3  {
4   bool amountChanged=bid_priv(msg.sender, value);
5   bid_callback(amountChanged);
6  }
7  function bid_callback(bool amountChanged) internal
8  {
9   if (amountChanged) {
10   // Increment bidCounter only for a new bid
11   bidCounter++;
12  }
13 }
14 function bid_priv(address user, uint64 value) internal
         returns (bool) {
15  uint64 existingBid = bids[msg.sender];
16  bool amountChanged = false;
17  if (existingBid >0) {
18   uint64 balanceBefore = tokenContract.balanceOf(
         address(this));
19   ...;
20   bids[msg.sender] = newBid;
21   amountChanged = true; // buggy assignment
22  }
23  else {
24   uint64 balanceBefore = tokenContract.balanceOf(
         address(this));
25   ...;
26   bids[msg.sender] = sentBalance;
27   amountChanged = true;
28  }
29  uint64 currentBid = bids[msg.sender];
30  if (highestBid == 0) {
31   highestBid = currentBid;
32  } else {
33   bool isNewWinner = highestBid < currentBid;
34   if (isNewWinner)
35   highestBid = currentBid;
36  }
37  return amountChanged;
38 }
```

Listing 2. Paritioned code by PARTITIONGPT.

contract, user bids stored in the data variable `existingBid` and the current highest bid `highestBid` are labeled as the sensitive data variables. To partition this function, PARTITIONGPT first performs taint analysis (see Algorithm 1 of Section III-C) to identify all the sensitive statements. As shown in Listing 1, the orange code blocks ②, ③, ④, ⑥, and ⑦ encompass all the sensitive statements while the other statements, e.g., the invocation to modifier `onlyBeforeEnd` (① of Section III-A) that checks if the auction ends and the increment of `bidCounter` (⑤ of Section III-A), belong to non-sensitive statements. Next, we perform slicing according to the aforementioned sensitive statements (see Section III-D), leading to two program slices. The privileged slice can be formalized as ① ② **if** ③ **then** ④ **else** ⑥ **fi** ⑦, while the normal slice is ① ② **if** ③ **then** {} **else** ⑤ **fi**. The two slices preserve the execution integrity for either sensitive and non-sensitive statements, but are coupled with each other since they share ① ② **if** ③. To decouple these, we leverage LLM's in-context learning by instantiating the generation template (see Fig. 3) with the original function code, identified sensitive statements, and computed normal and privileged slices. Finally, PARTITIONGPT employs LLM

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on May 15,2026 at 01:46:10 UTC from IEEE Xplore. Restrictions apply.

Generation/Repairing Prompt for Program Partitions

Suppose you are an expert developer for Solidity smart contracts. There is a code transformation task of smart contract function ([function code to be partitioned]) where two program slices, i.e., the normal and privileged slices ([normal slice] and [priviledged slice]), have been given.
In our slicing, slicing critera are a sequence of program statements that are labelled privileged ([privileged statements]). Your job is to transform the original contract function to a new variant encompassing these two program slices. The new function variant MUST be functionally equivalent with the original one.

[function code to be partitioned]: {func_code}
[privileged statements]: {privilege_stmts}
[normal slice]: {slice_normal}
[priviledged slice]: {slice_priv}

Bad partition output: // default is null
[one unsecure partition]: {unsecure_partition_result}
[explanation]: {unsecure_reason}

Please STRICTLY follow the below actions step by step:
1. MUST identify all the privilege statements including conditional checks shared between the two program slices.
2. MUST base on the provided privileged and normal slice for creating new sub functions. Privileged slice-based sub function in the form of “XXX_priv“ contains all the identified privileged statements. If priviledged functions need to yield return value, there must be a normal callback function in the form of “XXX_callback“ to process the return value. If there are normal statements to execute after the priviledged sub function, there must be a normal callback function in the form of “XXX_callback“ to process the normal statements.
3. NOTE if modifier statements contain privileged statements, then modifier statements MUST be included in the privileged sub function.
4. TRY to reduce those normal, i.e., non-privileged, statements in privileged sub functions as many as possible.
5. All the resulting code MUST satisfy the grammar of Solidity programming language.

You MUST output all the result in plain text format.
Only output the transformed contract code, and avoid unnecessary text description.

Fig. 3. The prompt for generating/repairing program partitions.

to generate program partition (see Section III-E) as shown in Listing 2. This normal partition comprises two functions–the refactored `bid` and created `bid_callback` functions while the privileged partition includes only one function called `bid_priv`. Note that the refactored `bid` remains the entry function for users to trigger smart contract execution. As illustrated in Listing 2, to orchestrate the execution between the privileged part `bid_priv` and the normal part `bid_callback`, PARTITIONGPT introduces new temporary flag variable `amountChanged` (Section III-A) and set the value at Section III-A and Section III-A. This flag variable will be returned (Section III-A) and when amountChanged is true (Section III-A), `bidCounter` will increase (Section III-A). However, the value assignment at Section III-A (colored in orange) is wrong because users have already held a bid position that can be implied by the non-zero existing bid price, thus nonequivalent with the pre-partition function in Listing 1. This indicates that although LLM commands powerful in-context learning capability, precise semantic understanding may be challenging and external verification tools (see Section IV) are essential to generate formal guarantee of correctness.

C. Identification of Sensitive Functions

Identifying critical functions is a key step in PARTITIONGPT's workflow, as it ensures the accurate isolation of sensitive statements. Developers manually specify privacy annotations, marking sensitive data variables that are considered privileged [60]. Developers can identify them through domain expertise, security auditing guidelines, or static analysis tools. This manual step is minimal in effort, as such variables are typically well-documented in contract interfaces or known from prior security incidents. These annotations guide the taint analysis and subsequent partitioning process, ensuring a focus on protecting critical data. Moreover, program dependence graphs (PDGs) are constructed to capture the dependencies between statements in smart contracts. The PDGs comprise control dependencies, representing the flow of control, and data dependencies, which highlight interactions between variables. The construction of PDGs is straightforward and

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on May 15,2026 at 01:46:10 UTC from IEEE Xplore. Restrictions apply.

quite standard, and readers could refer to [61] in which PDG construction is detailed for program partitioning. Algorithm 1 shows the taint analysis algorithm used to identify sensitive functions and sensitive program statement nodes. Algorithm 1 takes the user-provided sensitive state variables and constructed PDGs as input. We perform forward analysis to recognize all the sink nodes (Algorithm 1 to Algorithm 1) by enumerating each sensitive variable and each node of every PDG for different functions. We use isDependent to indicate the data dependencies between any two variables along the PDG, while for each node, *readVars*, *writeVars*, and *rvVars* represent the read variables, the written variables, and their combinations, respectively. When the sensitive variable has a data flow to a variable that the node reads or writes (Algorithm 1), we mark it as a sink node (Algorithm 1) and sensitive functions will be updated accordingly (Algorithm 1). Additionally, we perform backward analysis to recognize all the source nodes (Algorithm 1 to Algorithm 1) by revisiting all the variables that sink nodes read and the PDGs for different functions. When a node has a data flow to a variable that the sink node reads (Algorithm 1), we mark it as a source node (Algorithm 1) and sensitive functions will be updated accordingly (Algorithm 1).

Note we also have some limitations for the range of sensitive data variables. Solidity smart contracts could have composite data types like structure of which some member variables may be sensitive. To eliminate the complexity of data type splitting, we also label the corresponding composite data variables as sensitive, although it may result in a slightly larger set of sensitive operations.

D. Program Slicing and Partitioning

To employ the in-context learning capability of LLMs, PARTITIONGPT necessitates that contract functions are sliced based on privileged, i.e., sensitive statements, in order to generate high-quality program partitions that probably preserve the confidentiality and semantic integrity of original code.

Here, we formulate program slicing process and the constraints of slicing-based partition.

Definition 1: **Program Slicing.** Given a contract function $f = (\mathcal{S},\ \preceq_{control},\ \preceq_{data})$, $\mathcal{S}$ indicates the set of all program statement nodes. $\mathcal{S} = \{entry,\ \cdots,\ exit\}$ includes the input-related entry point and return-related exit point(s) of function. The two relations $\preceq_{control}$ and $\preceq_{data}$ represent the partial order between nodes in terms of control and data flow, respectively. For instance, $\exists\, a, b \in \mathcal{S}, a \preceq_{control} b$ delineates that a is control-dependent on b, while $a \preceq_{data} b$ implies that a is data-dependent on b. Note we assume $\forall a,\ a \preceq_{control/data} a$ always holds. Let Δ be the privileged nodes related to operations on sensitive data variables, and $\Delta \subseteq \mathcal{S}$. The program slice for privileged nodes can be defined as:

$$
\begin{aligned}
f \downharpoonright \Delta = (\mathcal{S}',\ \preceq_{control},\ \preceq_{data}) \quad w.r.t. \\
\mathcal{S}' = \{a \,|\forall\, a \in \mathcal{S}, \exists\, b \in \Delta\ \ (b \preceq_{control/data} a)\}
\end{aligned}
$$

where the program slice for non-privileged nodes $\overline{\Delta} = \mathcal{S} \setminus \Delta$, i.e., $f \downharpoonright \overline{\Delta}$, is defined similarly.

Definition 2: **Slicing-based Program Partitioning.** We partition f into two parts denoted as f'_{public} and f'_{priv}. To orchestrate the execution between the two function units, a special statement node, denoted as `priv_invoke` is added to f'_{public} that will trigger the execution of f'_{priv}. For simplicity, let $\mathcal{S}_{public}$ be the statement nodes of f'_{public} and $\mathcal{S}_{priv}$ for f'_{priv}. Note `special_invoke` $\in \mathcal{S}_{public}$. The program partitioning problem can be abstracted and defined as:

$$\forall a \in \mathcal{S}_{public},\ a \in \overline{\Delta} \tag{1}$$

$$\forall a, b \in \mathcal{S}_{priv},\ (a \preceq b)_{f'_{priv}} \implies (a \preceq b)_{f \downharpoonright \Delta} \tag{2}$$

$$\forall a, b \in \mathcal{S}_{public},\ (a \preceq b)_{f'_{public}} \implies (a \preceq b)_{f \downharpoonright \overline{\Delta}} \tag{3}$$

$$
\begin{aligned}
&\forall a \in \mathcal{S}_{public}, b \in \mathcal{S}_{priv}, \\
&(a \preceq b)_f \implies (a \preceq \texttt{priv_invoke})_{f'_{public}} \wedge (exit \preceq b)_{f'_{priv}} \\
&\textbf{and} \\
&(b \preceq a)_f \implies (\texttt{priv_invoke} \preceq a)_{f'_{public}} \wedge (b \preceq entry)_{f'_{priv}}
\end{aligned} \tag{4}
$$

where $\preceq$ represents $\{\preceq_{control}, \preceq_{data}\}$, $(\cdot \preceq \cdot)_x$ refers to the partial order of control or data flow for a given function/slice x.

Equation (1) expresses the security constraints where privileged statement nodes cannot be included in the public part f'_{public}. Equation (2) and Equation (3) assure the local integrity of control and data flow in f'_{public} and f'_{priv}, respectively. In contrast, Equation (4) examines the inter-procedure data or control flow integrity between f'_{public} and f'_{priv}. During program partitioning, we highlight that the data flow from f'_{public} to f'_{priv} permits only the user-provided parameters.

Wu et al. [62] have shown that the fine-grained program partitioning is NP-hard because it can be translated into the multi-terminal cut problem, which is a typical NP-hard problem. With an impressive capability of the in-context learning, LLMs can adapt to diverse coding styles and complex contexts, offering more nuanced and context-aware suggestions [63]. In this work, we leverage the capability of in-context learning of LLMs to effectively search for valid program partitions.

E. LLM-Driven Fine-Grained Partitioning

The process of fine-grained program partitioning in PARTITIONGPT leverages the power of LLMs to transform smart contract functions into securely partitioned variants. Using a carefully designed prompt (Fig. 3), PARTITIONGPT guides the LLM by preprocessing a function into two slices: the normal slice, containing non-sensitive statements, and the privileged slice, encompassing operations related to sensitive data variables. PARTITIONGPT's partitioning ensures that privileged operations are isolated from normal execution, creating a secure and modular structure within the smart contract.

The LLM performs this Solidity-to-Solidity transformation by strictly adhering to the guidelines provided in the prompt. In the partition result, privileged partition encapsulates privileged operations in a dedicated function (e.g., XXX_priv). For non-privileged partition, while refactoring the entry function, necessary callbacks (e.g., XXX_callback) are also introduced to handle return values or continue execution of normal statements

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on May 15,2026 at 01:46:10 UTC from IEEE Xplore. Restrictions apply.

for the purpose of higher modularity. Modifier statements[3] that include privileged operations will be incorporated into the privileged partition. By doing so, the LLM minimizes the inclusion of non-privileged statements in the privileged partition, ensuring a clear separation of concerns and enhancing security. Note the privileged and non-privileged partition by PARTITIONGPT communicates with each other through function calls (e.g. Listing 2). Additionally, we also provide some (currently two) human-written program partitions for contract functions as the seed examples to direct the LLM for program partitioning. While these examples may be limited, we argue that our preliminary experiment found that without few examples, the resulting program partitions often deviate from the aforementioned structure requirements listed in the prompt.

In cases where the generated partition does not meet security requirements, the LLM undertakes iterative repairs, which is discussed in the next section. The prompt is updated to include the insecure partition and a detailed explanation of its shortcomings, enabling the LLM to refine its output. This iterative process continues until a compilable and secure partition is produced. Through this LLM-driven approach, PARTITIONGPT achieves precise and reliable partitioning, ensuring that smart contracts are both robust and resistant to data leakage risks.

Nevertheless, inaccuracy could exist in some partition results. To mitigate this problem, for each subject function, PARTITIONGPT attempts to generate up to 10 partitions, where for each output code, PARTITIONGPT makes less than 10 tries to revise compilation error if available. The resulting partitions will be ranked and selected to represent the *appropriate* program partitions that developers are interested in. We will discuss the ranking process in Section III-G and illustrate one partition case in Listing 2 of Section III-B.

F. Revising and Repairing Program Partitions

The partition results by PARTITIONGPT may not be compilable because while partition generation seems straightforward for LLMs, it suffers from innate randomness to some extent. Following the practice [64], we leverage compiler feedback to revise the subject code.

The grammar-fix prompt shown in Fig. 4 is designed to leverage the expertise of LLMs to correct syntax errors in Solidity smart contract code, ensuring the output is grammatically valid while preserving the original logic and functionality. The prompt explicitly provides the incorrect code and corresponding compiler error messages, guiding the LLM to focus on specific issues without altering the program's intended behavior. By emphasizing strict adherence to Solidity grammar and requiring output in plain text without unnecessary explanations, the prompt ensures that the resulting code is concise, accurate, and ready for further validation.

The compilable partition code will be validated to determine if the partition is secure. Recall that the correctness of program partitioning can be verified by the four equations (c.f. Equation (1) to (4) of Definition 2). In practicality, Equation (2) to (4) are non-trivial to verify since LLMs could slightly modify original statement nodes for better clarity where new temporary variables and its related statements could be added, for which we leave such validation for equivalence checking in Section IV to derive robust guarantee. Fortunately, the security constraints expressed in Equation (1) can be easily checked by syntactically examining if the public part of partition result contains any privileged statement node that has data flow into or out of the given sensitive data variables. With this insight, we devise an effective detection rule based on Equation (1) to discover insecure partitions or obtain *likely* secure partitions. To repair the insecure partitions (c.f. Fig. 3), PARTITIONGPT will regenerate new program partitions until they satisfy the security constraints.

G. Ranking the Top-K Appropriate Program Partitions

Since optimal program partitions are usually subjective and hard to define, to avoid human bias and minimize manual efforts, we leverage a weighted selection algorithm to select the appropriate program partitions

Specifically, we establish a fitness function to evaluate the candidates by considering the following factors:

- $X(f, f')$: Edit distance between function f and its partition result f', which is computed as the normalized token-level Levenshtein distance between the original function f and the partitioned function f'. We use 1- $X(f, f')$ to measure the editing efforts.
- $Y_{codebase}(f', f'_{priv})$: The ratio of codebase size of privileged part f'_{priv} —the trusted codebase (TCB), compared to the whole partition result f'. We use 1-$Y_{codebase}(f', f'_{priv})$ to measure the extent to which TCB is minimized.
- $Y_{codebase}(f'_{priv}, \Delta)$: The ratio of codebase size of privileged statements Δ compared to the whole privileged part f'_{priv}. A higher ratio indicates a finer-grained TCB partitioning, which suggests the resulting TCB contains fewer non-privileged statements.

Given an unknown code f, we score f' using a weighted algorithm as below.

$$\begin{aligned} Score(f, f') = {} & \alpha \times (1 - X(f, f')) + \beta \\ & \times (1 - Y_{codebase}(f', f'_{priv})) \\ & + \gamma \times Y_{codebase}(f'_{priv}, \Delta) \end{aligned}$$

where α, β, γ are coefficients and $\alpha + \beta + \gamma = 1$. The intuition is that a good partition should introduce fewer edits while contributing to a highly minimized TCB with a finer-grained partitioning.

Let $\hat{f}$ be the human-written program partition result of f. To tune these coefficients, we train a linear regression model by approximating actual score $\hat{Score}(f, \hat{f})$ that is computed based on their text embedding similarity. We have conducted a primitive experiment on 1,267 program partitions generated by PARTITIONGPT, which are then randomly divided into a training set (70%) and a testing set (30%) through the established API *train_test_split* of the *sklearn* library. Note these partitions are distinct from the evaluated partitions appearing in Table I of

[3] In Solidity smart contracts, modifiers are often used to restrict user access of functions.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on May 15,2026 at 01:46:10 UTC from IEEE Xplore. Restrictions apply.

Grammar-Fix Prompt for Program Partitions

You are an expert Solidity developer. Your task is to fix grammar errors ([compiler error message]) in the given Solidity smart contract code ([incorrect code]) while ensuring the logic and functionality remain intact.

[incorrect code]:{input_code}
[compiler error message]:{error_msg}

All the resulting code MUST satisfy the grammar of Solidity programming language.
MUST Output only the Fixed Code: Provide the corrected Solidity code in proper format, and Avoid unnecessary text description.

Fig. 4. The prompt for fixing grammar errors of program partitions.

TABLE I
THE EXPERIMENT RESULTS ON PARTITION GENERATION UNDER THE SETTING TOP-3. #HIT = TP ∩ #SENSITIVE FUNC, *PRECISION* = $\frac{\text{TP}}{\text{TP+FP}}$, AND *RECALL* = $\frac{\#\text{HIT}}{\#\text{SENSITIVE FUNC}}$ WHILE $F_1 = \frac{2\times\text{PRECISION}\times\text{RECALL}}{\text{PRECISION}+\text{RECALL}}$

Contract	Secrete data	LoC	#Func	#Sensitive Func	#Partition	TP	FP	#Hit	Precision	Recall	F_1
ConfidentialID	identities	164	17	8	67	19	5	7	0.79	0.88	0.83
ConfidentialERC20	balances, allowances	117	9	5	42	15	0	5	1.00	1.00	1.00
TokenizedAssets	assets	21	3	3	30	9	0	3	1.00	1.00	1.00
EncryptedERC20	balances, allowances	109	16	6	60	13	5	5	0.72	0.83	0.77
DarkPool	orders, balances	84	8	7	50	9	6	3	0.60	0.43	0.50
IdentityRegistry	identities	167	17	8	73	19	5	7	0.79	0.88	0.83
BlindAuction	bids	190	13	5	45	8	7	3	0.53	0.60	0.56
ConfidentialAuction	bids	188	14	6	57	12	6	4	0.67	0.67	0.67
Battleship	player1/2-Board	78	3	2	13	5	1	2	0.83	1.00	0.91
VickreyAuction	bids	232	9	5	49	9	6	3	0.60	0.60	0.60
Comp	balances, allowances	136	9	7	62	15	5	5	0.75	0.71	0.73
GovernorZama	proposals	203	17	10	90	18	12	6	0.60	0.60	0.60
Suffragium	_votes	155	21	6	57	18	0	6	1.00	1.00	1.00
AuctionInstance	Biddinglist	149	8	3	18	0	0	0	NA	NA	NA
Leaderboard	players	23	3	3	28	9	0	3	1.00	1.00	1.00
NFTExample	_tokenURIs	602	34	2	19	6	0	2	1.00	1.00	1.00
EncryptedFunds	SupplyPerToken, etc.	168	16	6	60	16	2	6	0.89	1.00	0.94
CipherBomb	cards, roles	280	16	7	57	15	3	5	0.83	0.71	0.77
Overall		170	233	99	877	215	63	75	0.80	0.82	0.81

Section V. The results show α: 1.179, β: 0.373, and γ: -0.553 are aligned to human-written partitioned data. As a result, all the program partitions will be sorted in descending order, and we believe that the program partitions of higher rank are likely to be the high-quality program partitions.

IV. EQUIVALENCE CHECKING

The correctness of the resulting compilable and likely secure program partitions should be formally *verified* against the original subject code. To the best that we know, only one proprietary formal verification tool[4] provided by Certora [26] is able to perform equivalence checking between smart contract functions. However, their tool is closed-source and limited to only the comparison between two "pure" functions where a pure function cannot read and alter data variables of smart contracts[5].

To address this problem, in this work, we develop a dedicated equivalence checker for smart contracts that verify the correctness of program partitions. Our equivalence checker comprise two steps. First, we perform symbolic execution of any two different smart contract functions to gather all the possible execution paths. Second, we apply two essential correctness criteria, focusing on the consistency of state changes and the accuracy of returned values, to evaluate the equivalence between the execution results of the two functions.

Note that the detector is responsible for security validation, particularly ensuring that the sensitive partition adheres to the

[4]https://docs.certora.com/en/latest/docs/equiv-check/index.html

[5]https://solidity-by-example.org/view-and-pure-functions/

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on May 15,2026 at 01:46:10 UTC from IEEE Xplore. Restrictions apply.

security property formalized in Equation (1). This check is lightweight, focusing on detecting insecure information flows between the privileged and normal slices. In contrast, the dedicated prover proposed in this section is responsible for functional correctness verification, i.e., checking whether the partitioned contract is semantically equivalent to the original contract. This is a semantic equivalence checker based on symbolic execution and constraint solving, ensuring that no observable behaviors of the original program are altered during partitioning.

a) Symbolic Execution: The execution of smart contracts relies heavily on persistent states stored on the blockchain, transaction environment data, and specific contract statements. These persistent states consist of contract state variables, while transaction messages include details such as the caller, callee, method invoked, and block timestamp. To model this execution, each contract statement is represented as a Hoare triple $\delta; s; \delta'$, capturing the program states before (δ) and after (δ') execution. Unlike traditional programs, smart contracts do not crash; unexpected behaviors trigger transaction reversions, leaving the contract state unchanged. Although such reversions can impact availability (e.g., denial of service), they do not generally compromise safety and are thus excluded from our analysis. To conduct the strongest postcondition analysis, we employ a source-level symbolic execution tailored to the Solidity language, which implements comprehensive small-step semantics [65]. For more details, readers can refer to the previous works [25], [66].

b) Equivalence Criteria: In general, equivalence between two programs is undecidable. Nevertheless, in the field of program partitioning, typically new functionality is not allowed to be added so that we can verify the functional equivalence between pre-partition and post-partition programs. The equivalence is defined by the following two properties:

- **State Change Consistency**. Given the same input, the state change of the pre-partition function is the same as that of the post-partition function.
- **Return Value Consistency**. Given the same input, the return value of the pre-partition function is the same as that of the post-partition function.

Note that we do not include the consistency property about transaction reversion because if a function execution is rolled back due to transaction reversion, the contract state remains unchanged, which can be indirectly implied by state change consistency.

Algorithm 2 shows the algorithm for checking state change consistency where the algorithm for checking return value consistency is similar and elided for saving space. To perform the checking for state change consistency, we assemble all the collected symbolic execution paths for two contract functions. Next, for each program, the evaluation of every state variable along all the execution paths will be conjunct and then be checked against the others using SAT solvers like Z3 [67]. We will raise a functionally non-equivalent report if the checking fails. The checking of return value consistency is similar, and we elide it to save space. When state change and return value consistency remain true, the correctness of the program partition is successfully verified.

Algorithm 2 Checking state change consistency

```
Require: 𝒱, the set of contract state variables.
         f, a pre-partition, i.e., original contract function.
         f′, a post-partition function of f.
Ensure: isEquivalent, a boolean flag indicating equivalence.
 1: let δ0 represent the symbolic values of variables in 𝒱.
 2: Φf ← SYMEXE(f, δ0)        ▷ Include all execution paths of f.
 3: Φf′ ← SYMEXE(f′, δ0)      ▷ Include all execution paths of f′.
 4: for v ∈ 𝒱 do
 5:     p ← False
 6:     q ← False
 7:     for φ ∈ Φf do
 8:         p ← p ∨ (φ ⟹ v = EVAL(φ, v))
 9:     for φ ∈ Φf′ do
10:         q ← q ∨ (φ ⟹ v = EVAL(φ, v))
11:     if SAT(¬(p ≡ q)) then
12:         Report "f and f′ are not functionally equivalent on v"
13:         return False
14: return True
```

Consider the original function bid of BlindAuction in Listing 1 and its partitioned variant in Listing 2 as inputs to equivalence checking. Notice that `bidCounter` is a state variable. For simplicity, the SMT formula related the state change regarding `bidCounter` in Listing 1 could be abstracted as

$$bidCounter{=}ITE(bids[msg.sender] > 0, old(bidCounter), old(bidCounter) + 1) \tag{5}$$

where **ITE** (*cond, t, e*) represents an if-then-else expression that evaluates a condition **cond** and returns two other expressions: t or e based on whether the condition is true or false, and **old** (x) represents the old value of a state variable x. However, for Listing 2, inlining the symbolic execution of `bid_priv` and `bid_callback` will produce a conjunction of formulae abstracted as

$$bidCounter = ITE(amountChanged, old(bidCounter){+}1, old(bidCounter)) \wedge amountChanged = ITE(bids[msg.sender] > 0, true, true) \tag{6}$$

Furthermore, after reduction of Equation (6), the resulting formula is $bidCounter = old(bidCounter) + 1$, of which the value assignment of `bidCounter` differs from that of Equation (5). Therefore, Listing 2 is not equivalent with Listing 1.

We highlight that we leverage modular verification to prove the equivalence. During modular verification, we lift all state constraints by making all state variables symbolic. Correctness can be safely ensured when the equivalence checking properties hold accordingly.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on May 15,2026 at 01:46:10 UTC from IEEE Xplore. Restrictions apply.

V. Evaluation

A. Implementation

We implemented our approach in PartitionGPT in around 4,000 lines of Python for partition generation and 500 lines of C++ for equivalence verification built on SolSEE [66]. We used Z3 solver [67], version 4.13.0, to check equivalence relationship between the original and partitioned function versions. PartitionGPT is empowered by *GPT-4o mini* developed by OpenAI, where its default model setting is reused. We also employed the embedding model *text-embedding-ada-002* developed by OpenAI to score LLM-written program partitions compared to human-written ones in terms of embedding similarity. We used the Levenshtein distance to measure the edit distance between the original functions and its partitioned output. Furthermore, we used Slither [68], version 0.10.4, for PDG construction and taint analysis for Solidity smart contracts.

B. Research Question

In the evaluation, we aim to answer the following research questions.

- **RQ1**. How accurately does PartitionGPT generate fine-grained program partitions for smart contracts?
- **RQ2**. How useful is PartitionGPT to mitigate real-world smart contracts from attacks?
- **RQ3**. What is the run-time gas overhead of PartitionGPT ?
- **RQ4**. How do different base LLMs affect the performance of PartitionGPT ?

a) Benchmark and ground truth: To answer RQ1, we collected real-world confidential smart contracts tagged with sensitive annotations from previous studies on fully homomorphic encryption-based solution fhEVM [69] and MPC-based computation-based solution COTI [70]. Specifically, we initially obtained 10 cases from fhevm-examples [71], 6 cases from COTI-examples [72], and additionally crawled from GitHub 6 smart contract applications participating in the competition using the fhEVM. We manually analyzed these contracts, and excluded 4 cases involving only the encryption of input data and the use of TEE-generated random number because they do not apply to program partitioning. Finally, we built a confidential smart contract benchmark encompassing 18 smart contracts with sensitive annotations as shown in Table I. Specifically, there are 99 sensitive contract functions related to the sensitive data variables. To obtain the ranking coefficients (c.f. Section III-G), we manually partitioned these contract functions to construct a human-written dataset. To avoid manual mistakes, we validate program partitions by performing equivalence checking using the proposed formal verification tool.

To answer RQ2, we searched all the price manipulation attacks recorded in the well-studied DeFi hack repository [73] as of September 2024. There are 63 incidents labeled with price manipulation attacks. We investigate the root cause of the attack, and found the root cause analysis reports for 25 cases are missing, and 15 of the rest cases are actually due to other vulnerability issues such as permission bugs or not open source. For the remaining 23 cases, 13 cases were attacked because of the use of vulnerable API *getAmmountIn/getAmmountOut* by Uniswap V2, while the rest cases are caused by other customized API uses. To avoid bias in the evaluation, we selected two case of *getAmmountIn/getAmmountOut*, i.e., Nmb-platform and SellToken, and select five out of the remaining 11 cases in the manipulation-related victim contract benchmark. Beside manipulation attacks recorded in DeFi hack repository, we include two reported randomness manipulation attacks[6]. Finally, to answer RQ2, we curated 9 attacks, leading to a total loss of about 25 million dollars.

b) Experiment setup: All the experiments are conducted on a server computer equipped with Ubuntu 22.04.5 LTS, 504 GB RAM, and 95 AMD 7643 Cores. We used the commercial OpenAI API to access *GPT-4o mini* and *text-embedding-ada-002*. Moreover, we used the advanced open source LLMs including Gemma2:27b, Llama3.1:8b, Qwen2.5:32b supported by Ollama[7] for the sensitivity study in RQ4. We allowed the symbolic execution unrolls up to five times for loop statements and the overall time budget for the equivalence checking was capped at ten minutes. The evaluation artifact is available on https://github.com/academic-starter/PartitionGPT.

C. RQ1: Partition Generation

We evaluated PartitionGPT on the aforementioned 18 confidential smart contracts encompassing 99 sensitive functions. Table I shows the experiment results under the Top-3 setting. The first five columns on the left demonstrate the name, sensitive data variables, line of codes, number of public functions, and number of sensitive functions of smart contracts, respectively. The middle two columns present the number of sensitive functions identified by PartitionGPT and the number of generated program partitions for those identified sensitive functions that match with the ground truth. Note that incorrect partitions failing equivalence checks are discarded by PartitionGPT (see Fig. 2). The false positives (FP) reported in Table I refer to a separate measurement context–when evaluating the accuracy of LLM inference (not after the prover filtering). The remaining columns demonstrate the verification results of the resulting partitions after equivalence checking. A true positive (TP) and false positive (FP) refer to a program partition result by LLM that passed and failed the equivalence checking, respectively. #Hit measures the number of sensitive functions that have at least one true positive. Accordingly, success rate, i.e., $Recall = \frac{\#\text{Hit}}{\#\text{Sensitive Func}}$, refers to the proportion of sensitive functions that have true positives.

Table I clearly shows that PartitionGPT is able to generate a reasonably high accuracy of program partitioning for smart contracts. Overall, PartitionGPT is able to generate 877 program partitions for 99 ground truth functions out of which 75 functions have been successfully partitioned, achieving a recall

[6]https://owasp.org/www-project-smart-contract-top-10/2023/en/src/SC08-insecure-randomness.html

[7]https://github.com/ollama/ollama

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on May 15,2026 at 01:46:10 UTC from IEEE Xplore. Restrictions apply.

of 0.82, with a precision of 0.80. Particularly, PARTITIONGPT failed to perform equivalence checking for AuctionInstance due to timeout. This is because AuctionInstance has nested loops in its sensitive functions, and although we cap the loop iteration count to five during symbolic execution, there remains a state explosion problem.

PARTITIONGPT currently does not support declassification annotation in the identification of sensitive functions. For instance, PARTITIONGPT flagged a declassified function `open` of `CipherBomb` as a sensitive function because this function can write a data variable `turnDealNeeded` that will flow to sensitive data `cards` and `roles` within `checkDeal` function. However, the checkDeal explicitly declares turnDealNeeded as a declassified variable through a decryption statement “turnDealNeeded = !TFHE.decrypt(...)”, where TFHE is a homomorphic encryption library of fhEVM [69]. Although practical in application development, declassification could have information exposure problem [74]. Thus, we leave declassification support as future work.

As shown in Table I, for 24 sensitive functions, PARTITIONGPT reports that no functionally equivalent partition exists. In practice, such functions could be treated as non-partitionable trusted cores and protected using stronger defenses, such as deployment of the whole function within trusted execution environments. Rather than indicating a limitation, these cases highlight security-critical logic whose tight coupling fundamentally resists fine-grained separation.

We also investigate the impact of different top-K settings. Fig. 5 demonstrates the averaged precision, recall, F_1 -score by PARTITIONGPT when K ranges from 1 to 5. Fig. 5 delineates that precision and recall fluctuate a bit between Top-1 and Top-3, and then remains relatively stable. Additionally, we also measure the TCB minimization performance, i.e., $Y_{codebase}(f', f'_{priv})$ (see Section III-G). In contrast to those coarse-grained approaches [50], [61], [62] that take the whole sensitive functions as TCBs with none of function splitting, our experiments indicate that PARTITIONGPT achieves a TCB reduction rate between 26.4% and 27.3% within the five aforementioned Top-K settings. Therefore, we believe that PARTITIONGPT is able to generate fine-grained partitions that decouple sensitive operations with normal operations. For practicality, we suggest using Top-3 as the default setting for PARTITIONGPT.

Effectiveness of Equivalence Checking. In the 877 program partitions generated by LLM. PARTITIONGPT reported 675 true positives that passed equivalence checking, and 184 false positives that failed equivalence checking, leading to an overall precision of 0.78. Additionally, there are 18 cases, i.e., `AuctionInstance`, that exceed the time budget. There are four causes of these false positives. We found 76 cases failing due to the state change inequivalence, 38 cases failing due to the unmatched return value, and 24 cases are caused by incompatible partition results, e.g., having inputs different from the original function. The remaining 48 cases are caused by the internal implementation issues, e.g., incorrect Z3 array conversion from complicated structure variables. Thereby, excluding 64 cases caused by timeouts and implementation issues, PARTITIONGPT

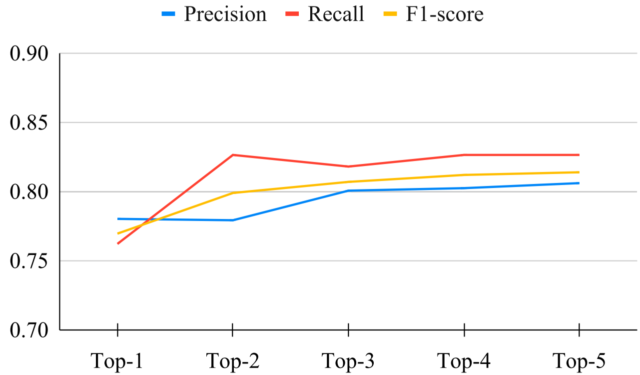


Fig. 5. The impact of Top-K selection.

is effective in verifying 813 (92.7%) cases. Besides, we noticed that since our equivalence checker is conservative and some reported false positives may not be true. For instance, we found for `BlindAuction`, PARTITIONGPT generates 5 partitions for its `bid` function (c.f., Listing 1) which involve complicated external contract calls beyond the capability of the developed dedicated equivalence checker. Although PARTITIONGPT reported all the five partitions are false positives, we manually checked and confirmed that three of them are actually true positives.

Effectiveness of Parameter Settings. To investigate how the tuned ranking coefficients in Section III-G affect the performance of PARTITIONGPT, we conducted an ablation study to compare PARTITIONGPT with its variant PARTITIONGPT *w/o ranking* that randomly picks partitioning results. To avoid bias, we have experimented on PARTITIONGPT *w/o ranking* 20 times and we used the averaged recall, precision, and F_1 score as its result in Table II. Table II shows the comparison results under different Top-K settings. We performed Mann Whitney U-test to measure how significantly PARTITIONGPT differ from PARTITIONGPT *w/o ranking*. Additionally, we conducted Vargha and Delaney $\hat{A}_{12}$ statistical test to determine the extent to which PARTITIONGPT outperforms PARTITIONGPT *w/o ranking*. Although the performance difference seems not significant under the first two settings, Table II indicates that PARTITIONGPT achieves a higher precision and F_1 score than PARTITIONGPT *w/o ranking* under Top-3, Top-4, and Top-5 settings, whose averaged p-value is 0.028, demonstrating a statistically significant difference, which is smaller than or close to a significance level of 0.05. Moreover, their averaged $\hat{A}_{12}$ is 0.866, which corresponds to a large effect size ($\hat{A}_{12}$ is larger than 0.71). Therefore, the tuned ranking parameters are effective and useful for smart contract partitioning.

Answer to RQ1: PARTITIONGPT is able to generate secure program partitions for smart contracts with a reasonably high accuracy. Under Top-3 setting, among 99 functions to partition, PARTITIONGPT gains a precision of 80% and a recall of 82% and reduces more than 26% code in TCB.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on May 15,2026 at 01:46:10 UTC from IEEE Xplore. Restrictions apply.

TABLE II
THE COMPARISON RESULTS BETWEEN PARTITIONGPT AND PARTITIONGPT *W/O RANKING*. NOTE PREC. INDICATES PRECISION

Tool	Top-1			Top-2			Top-3			Top-4			Top-5		
	Recall	**Prec.**	F_1	**Recall**	**Prec.**	F_1	**Recall**	**Prec.**	F_1	**Recall**	**Prec.**	F_1	**Recall**	**Prec.**	F1-score
PARTITIONGPT	0.76	0.78	0.77	0.80	0.78	**0.79**	0.82	**0.80**	**0.81**	0.83	**0.80**	**0.81**	0.83	**0.81**	**0.81**
PARTITIONGPT w/o ranking	0.78	0.79	0.78	0.82	0.78	0.80	0.83	0.78	0.80	0.83	0.79	0.80	0.83	0.79	0.80
p-value	0.0853	0.0853	0.0853	0	0.0853	0.0005	0	0	0.1477	0	0	0.0215	0	0	0
$\hat{A_{12}}$	0.35	0.35	0.35	0.5	0.35	0.2	0.05	0.9	0.625	0	1	0.7	0.5	1	0.975

D. RQ2: Application in Real-World Victim Contracts

We evaluated PARTITIONGPT on nine victim contracts vulnerable to price and randomness manipulation. Recall PARTITIONGPT is able to identify all the sensitive functions related to given sensitive data that we believe play pivotal role in the manipulation attack where attackers interfere with the sensitive data without any protection. In typical manipulation incidents, attackers first manipulate the sensitive data and then earn a profit. Most attacks like the well-studied price manipulation will incur at least two function calls where the first function call will alter sensitive data variables about liquidity, and the second function call will make a profit with the use of the manipulated data.

We compare PARTITIONGPT with the existing manipulation detection tools DefiTainter [22], DeFort [20], DefiRanger [21], and GPTScan [16]. Because DefiRanger is not open source, we take their reported results [21] in Table III. Different from DefiTainter and DeFort demanding function-related annotations, PARTITIONGPT needs developers to provide a set of annotations on sensitive data variables of smart contracts. We manually analyze the source code of smart contracts and identify security-critical related data variables as the sensitive data variables to protect. For instance, the data variables "_liability" and "_totalSupply" of BelugaDex are critical since they determine the amount of tokens to burn, which is vulnerable to arbitrary manipulation by attackers. Table III shows the comparison results. PARTITIONGPT is able to mitigate eight attacks by reducing the attack surface, followed by DeFiRanger and GPTScan. PARTITIONGPT failed to defend against **BelugaDex**'s attack because this attack arises from a flawed token withdrawal logic rather than manipulating the constant token exchange rate between two stable coins: USDT and USDCE. The withdrawal amount depends on the two statuses of the underlying asset token: *_liability* equaling to the sum of deposit and dividend, and *totalSupply* equaling to the sum of token distribution. In this exploit, the attacker deposited USDT tokens and then used swap function between USDT and USDCE to update asset liability. Consequently, the attacker spent less USDT asset tokens to withdraw the same original deposit tokens. Although PARTITIONGPT could hide *_liability* and *totalSupply* within a secure environment, the abovementioned attack vector still exists.

We detail how PARTITIONGPT mitigates the following attacks.

- **Nmbplatform** and **SellToken** use Uniswap V2-based liquidity pools to swap token A for another token B, but these pools are vulnerable to price manipulation because the reserves of two tokens within the pools are visible to the public. Therefore, malicious users can easily create a significant liquidity imbalance by depositing a certain amount of tokens borrowed using flashloan technique. PARTITIONGPT mitigates these attacks through marking the reserves of two tokens within the pools as sensitive and hiding all operations on these reserves in a privileged partition inside a secure environment.
- **Indexed Finance** uses a custom price model to calculate token exchange rate based on a critical data structure called *_records*. Because one of its fields named *denorm* is not affected by the change of the liquidity within the contract, attacker is able to manipulate the contract to forge a flawed token price that creates an arbitrage opportunity. PARTITIONGPT treated *_record* as sensitive data variables, and created privileged partition for the sensitive operations. Therefore, *denorm* of *_record* is invisible to malicious users, reducing the chance of gaining a profit.
- **Jimbo**'s price calculation relies on the complicated states of bins comprising *activeBin*, *triggerBin*, and so on (c.f. Section II-D). By observing and interfering with these bins, malicious users are able to initiate manipulation attacks. PARTITIONGPT places all sensitive operations related to these bins into privileged partition for executing within a secure environment, making bins' states always sealed to mitigate such manipulation.
- **NeverFall** and **BambooAI** are liquidity pool token contracts implementing ERC20 standard [75], where NeverFall permits users to sell and buy liquidity tokens, and BambooAI allows users to update the pool's parameters when performing token transfers. In both of the two attacks, malicious users manipulate a bookkeeping variable *balances* that manages the distribution of tokens among users and is responsible for the liquidity price calculation, through selling, buying, and transferring a certain amount of tokens. PARTITIONGPT defends against these attacks by isolating all operations related to *balances* in a privileged partition, making it hard for attackers to predicate the amount of tokens they need to carry out attacks.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on May 15,2026 at 01:46:10 UTC from IEEE Xplore. Restrictions apply.

TABLE III
THE EXPERIMENT RESULTS ON MANIPULATION ATTACK MITIGATION

Victim	Date	Loss	DefiTainter	DeFort	DefiRanger	GPTScan	PARTITIONGPT
Nmbplatform	14-Dec-2022	$76k	✓	✓	✓	✗	✓
SellToken	11-Jun-2023	$109k	✗	✗	✓	✗	✓
Indexed Finance	15-Oct-2021	$16M	✗	✗	✓	✗	✓
Jimbo	29-May-2023	$8M	✗	✗	NA	✓	✓
NeverFall	3-May-2023	$74K	✓	✗	✗	✗	✓
BambooAI	4-Jul-2023	$138K	✗	✗	✓	✓	✓
BelugaDex	13-Oct-2023	$175K	✗	✗	NA	✗	✗
Roast Football	5-Dec-2022	$8K	✗	✗	NA	✗	✓
FFIST	20-Jul-2023	$110K	✗	✗	NA	✓	✓
Overall			2	1	4	3	8

- **Roast Football** and **FFIST** are two contracts on BSC that use on-chain data for random number generation for lottery and token airdrop purpose, respectively. For instance, as shown in Listing 3, the lottery of Roast Football leverage a function *randMod* to generate random number (Section V-D) from the current block number and timestamp, user-given input *buyer*, and contract-managed data *_balances* recording token distributions (Section V-D). As the sensitive random seed *_balances* is known to the public, malicious users could easily manipulate the input with an elected buyer address to generate a desired random number that bypasses any of the branch constraints in this function. Considering *_balances* as sensitive data variable, PARTITIONGPT isolates all the data and operations (in orange box, Lines 2 and 3) about random number generation, thus disallowing attackers to predicate random number output. We also highlight that some approach [11], [76] could complain the use of block-related data like *block.number* and *block.timestamp* since they may be manipulated by blockchain miners, and such manipulation can be prevented through decentralized governance mechanisms such as proof of stakes, which is out of our research scope. FFIST generates random number with a previously airdropped address stored in the contract. The way PARTITIONGPT protects this generation is similar to *Roast Football*, and we elide it for saving space.

Therefore, we argue that PARTITIONGPT is able to effectively mitigate real-world manipulation attacks through secure program partitioning.

```
1  contract RFB{
2    mapping (address => uint256) _balances;
3    function randMod(address buyer,uint256 buyamount)
         internal returns(uint){
4      uint randnum = uint(keccak256(abi.encodePacked(block.
           number,block.timestamp,buyer,_balances[pair])));
5      uint256 buyBNBamount = buyamount.div(10**_decimals).
           mul(getPrice());
6      // increase nonce
7      if(randnum%(10000*luckyMultiplier) == 8888 &&
           buyBNBamount > (0.1 ether)){
8          distributor.withdrawDistributor(buyer, 79);
9          distributor.withdrawDistributor(
               marketingFeeReceiver,9);
10     }else if(randnum%(1000*luckyMultiplier) == 888){
11         ...
12     }else if(randnum%(100*luckyMultiplier) == 88){
13         ...
14     }else if(randnum%(10*luckyMultiplier) == 8){
15         ...
16     }
17     return randnum;
18   }
19 }
```

Listing 3. The *randMod* function of Roast Football.

Answer to RQ2: PARTITIONGPT is able to defend against eight of nine real-world manipulation attacks beyond the existing detection-only tools, underscoring the potential of secure program partitioning in preventing sensitive data leakage to mitigate manipulation attacks.

E. RQ3: Runtime Overhead

The partitions produced by PARTITIONGPT should be deployed to secure execution environment to protect sensitive functionality. Trusted execution environment (TEE) has been employed to protect the privacy of smart contracts [40], [41]. TEE is able to execute general instructions but with minimal time cost. There have been several TEE infrastructures designed for smart contracts such as open-source CCF [41] in which the whole Ethereum Virtual Machine is running at SGX enclave mode supporting TEE devices like Intel SGX.

To conduct the experiments, we select eight functions from two contracts listed in Fig. 6. The first four functions belong to `BlindAuction`, and they are responsible for the biding, querying and post-biding processing, while the latter four functions of `EncryptedERC20` represent the most common ERC20 functionalities. For each contract, it includes two executable instances: one for the public part deployed in a Ethereum test network powered by Ganache [77] of version 6.12.2, and the other for the privileged part deployed in the CCF network [41] with a simulated TEE environment. For the cross-chain communications between Ethereum-side and TEE-side instance, we replace the call statements, i.e., XXX_priv(), with message events logged in the normal test network, which will then be passed to a third-party router to trigger the execution of privileged function within the TEE-side instance. Also, we replace the call statements, i.e., XXX_callback(), with message events logged in the CCF network and then the third-party router processes the events to

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on May 15,2026 at 01:46:10 UTC from IEEE Xplore. Restrictions apply.

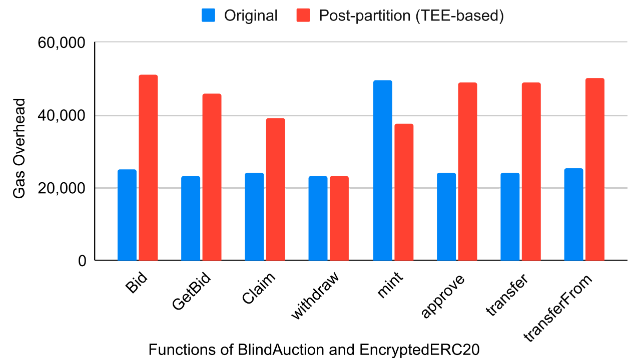


Fig. 6. The runtime overhead of deploying partitions.

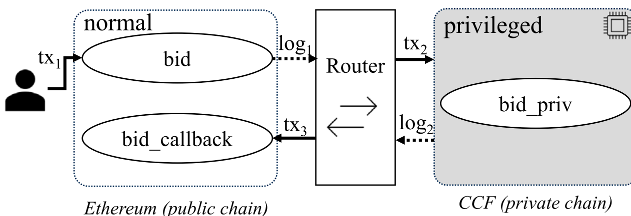


Fig. 7. Secure deployment demonstration of Listing 2.

trigger the callback function of the Ethereum-side instance. To orchestrate the communication, we formulate their call orders into scheduling policies. We highlight that the aforementioned substitution rules can be automated by using a few code transformation templates. Furthermore, we implemented the third-party router as a listener thread to automatically monitor new events from the normal test network and the CCF network and then deal with the events according to the abovementioned scheduling policies. Fig. 7 showcases the secure deployment of partitions in Listing 2. The user first sends a transaction to the entry function *bid*, and later the router will be in charge of scheduling two messages and two transactions from and to the public chain and private chain, respectively. To assess the performance, for each function, we manually crafted five test cases to execute and then collected their runtime gas overheads. We clarify that currently we do not add encryption and decryption methods for the data communicated between the Ethereum-side and TEE-side instance. Developers of smart contracts should be responsible for this particular setting. For instance, developers may need to exchange their keys through transactions or smart contracts where the data will be decoded in the TEE-side [49], [57].

Fig. 6 plots the runtime gas overhead of the original contract and the TEE-powered post-partition contract. Note that the execution of the TEE-side instance will not incur gas overhead for the CCF network. Apparently, after partitioning, six out of eight contract functions will take more gas overhead, increasing between 61% and 103%. The main reason is that these functions not only communicate message from the Ethereum-side instance with TEE-side instance but also deal with callback message from TEE-side instance to require Ethereum-side instance to proceed with normal operation execution, leading to two more transactions. In comparison, for the two functions: `withdraw` and `mint`, callback-related communications are not needed, thus reducing the gas consumption. Note each communication will take one transaction, and 21,000 gas is charged for any transaction on Ethereum as a "base fee" so that n transactions will need to consume no less than n $\times$ 21,000 gas. Therefore, we argue that although deploying the partitions into TEEs could incur more gas overhead, developers could still benefit in twofold. First, sensitive functions usually take a small proportion of all the public functions (c.f., 99 out of 233 in Table I), resulting in a moderate gas overhead increase for users. Second, the attack surface of smart contracts is largely minimized, and it will be difficult for attackers to manipulate the sensitive data variables within a secure environment like TEE.

Answer to RQ3: The runtime gas overhead is moderate by deploying partitions to a TEE-based blockchain infrastructure, where except for two cases, after partitioning by PARTITIONGPT, six of eight functions incur 61% to 101% more gas mostly paid for additional communication transactions.

F. RQ4: Sensitivity Study

To investigate the effectiveness of different base LLMs, we conducted a sensitivity study on partition generation for the aforementioned 18 confidential smart contracts (c.f. Table I) with four LLMs: (1) GPT-4o mini, (2) Gemma2:27b, (3) Llama3.1:8b, and (4) Qwen2.5:32b.

Table IV shows the comparison results. We evaluated LLMs on the overall number of generated partitions, true positives, false positives, and precision. Recall that we excluded AuctionInstance's partitions in the precision evaluation. Note #Partitions also includes partitions beyond function scope of the ground truth. It is evident that GPT-4o mini substantially outperforms over the rest LLMs. GPT-4o mini yields 877 partitions in total, achieving a precision score of 0.78, followed by Qwen2.5:32b having 758 partitions and precision of 0.75. Llama3.1:8b performs the worst, rendering the least number of partitions and the poorest precision (0.44). To have an in-depth analysis of such significant difference in the number of resulting partitions, we studied the capability of different LLMs in generating compilable program partitions for smart contracts, which plays a vital role in our approach. Fig. 8 depicts the distribution of times that different LLMs need to generate compilable partitions. Fig. 8 indicates that 83% partition candidates initially generated by GPT-4o mini are compilable, followed by Qwen2.5:32b having 70%. It is surprising that all the partition candidates generated by Llama3.1:8b must be fixed at least once (using ⑤ **Revise** of PARTITIONGPT). The main reason, we believe, may be that Llama3.1 has not been trained on Solidity smart contract datasets. Therefore, we recommend using the closed-source GPT-4o mini for efficiency and the open-source Qwen2.5:32b to facilitate smart contract partitioning with budget consideration.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on May 15,2026 at 01:46:10 UTC from IEEE Xplore. Restrictions apply.

TABLE IV
EFFECTIVENESS WITH DIFFERENT LLMS

LLM	#Partitions	TP	FP	Precision
GPT-4o mini	877	675	184	0.78
Gemma2:27b	621	436	149	0.75
llama3.1:8b	227	98	121	0.44
Qwen2.5:32b	758	476	157	0.75

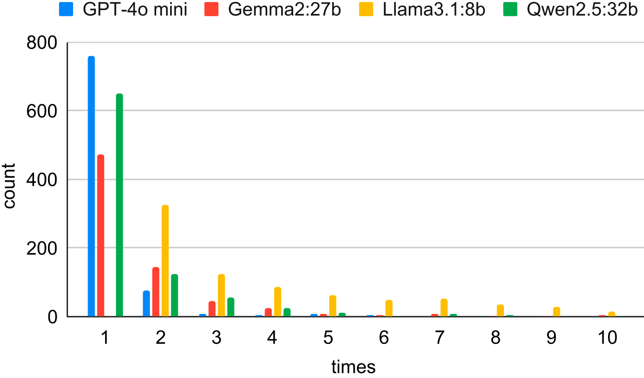


Fig. 8. The distribution of the times that different LLMs need to generate compliable smart contracts.

Answer to RQ4: The selection of base LLMs could affect the performance of PARTITIONGPT. The sensitivity study found that GPT-4o mini generates the largest number of partitions while achieving the highest precision of 0.78, followed by Qwen2.5:32b.

G. Threats to Validity

a) Lack of ground truth: We manually selected a set of annotated smart contracts and real-world smart contracts vulnerable to manipulation attacks for evaluating the effectiveness of PARTITIONGPT. To address this threat, we systematically crawled well-studied confidential smart contracts in which developers have explicitly labeled sensitive data, as well as victim smart contracts from known real-world attacks with root causes scrutinized by security experts.

b) External validity: Our findings in program partitioning may not generalize to other large language models. To mitigate this threat, we have evaluated PARTITIONGPT with state-of-the-art LLMs developed by four vendors, namely closed-source GPT-4o mini by OpenAI, open-source Llama3.1 by Meta, Qwen2.5 by Alibaba, and Gemma2 by Google.

VI. DISCUSSION

Given accurate privileged and normal slices, one could, in principle, design a deterministic algorithm to construct a partitioned smart contract variant by exhaustively restructuring control and data dependencies. However, such approaches typically require rigid transformation templates or incur combinatorial search over syntactically valid rewritings. PARTITIONGPT instead treats partition synthesis as a candidate generation problem and leverages LLMs to efficiently explore the space of functionally equivalent variants.

PARTITIONGPT integrates multiple program analysis and verification techniques, namely program slicing, syntax-level program repair, and symbolic execution, to support fine-grained partitioning of smart contracts. While these techniques may face scalability or precision challenges when applied to large-scale, general-purpose software systems, their adoption in PARTITIONGPT is a deliberate and principled design choice grounded in the unique characteristics of the smart contract domain.

Smart contracts differ fundamentally from traditional software systems in both structure and execution semantics. In practice, smart contracts are typically compact in size, have deterministic execution, and expose well-defined state variables and entry points. Control flow is relatively shallow, concurrency is limited or absent, and interactions with the external environment are mediated through explicit transaction boundaries. These properties substantially reduce the analysis complexity faced by program analysis techniques that are otherwise challenged in large and evolving codebases. As a result, techniques such as slicing and symbolic execution can be applied in a focused and tractable manner, making them well-suited for reasoning about smart contract partitioning.

PARTITIONGPT employs program slicing to isolate code regions that influence security-sensitive state variables. Rather than performing whole-program or arbitrary slicing, slicing criteria are explicitly derived from identified sensitive functions and state variables. This property-driven slicing significantly reduces irrelevant code and enables precise identification of the minimal logic required to enforce security isolation. In the smart contract setting, where data dependencies are easily computable and global state is limited, slicing provides an effective mechanism for separating sensitive and non-sensitive logic without introducing excessive noise.

Moreover, PARTITIONGPT is designed as a modular framework rather than a monolithic analysis pipeline. Each analysis component, including slicing, grammar repair, and symbolic reasoning, can be replaced or augmented with more advanced or domain-specific techniques as they become available. This modularity ensures that PARTITIONGPT remains extensible and adaptable to future advances in program analysis and smart contract security.

VII. RELATED WORK

A. Smart Contract Security

The detection of vulnerabilities in smart contracts has been a central focus of blockchain security research. Symbolic execution tools like Oyente [11], Manticore [23], and Mythril [24] pioneered the detection of critical vulnerabilities such as reentrancy, mishandled exceptions, and integer overflow/underflow. These tools systematically explore execution paths to identify potential exploits but are limited by path explosion and incomplete semantic coverage. Static analysis tools, such as Slither [78] and SmartCheck [13], leverage data-flow and control-flow

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on May 15,2026 at 01:46:10 UTC from IEEE Xplore. Restrictions apply.

analyses to identify a wide range of vulnerabilities, including bad coding practices and information-flow issues. Securify [79] and Ethainter [14] use rule-based pattern matching to detect vulnerabilities, while SmartScopy [15] introduces summary-based symbolic evaluation for attack synthesis. Dynamic analysis techniques, including fuzzing tools like ContractFuzzer [17], sFuzz [18], and Echidna [19], analyze runtime behaviors by generating test inputs to uncover exploitable bugs. However, these tools often suffer from limited state coverage and rely heavily on predefined oracles. Formal verification tools, such as KEVM [27] and Certora [26], as well as semi-automated tools like Echidna [19], require user-provided specifications, such as invariants or pre-/post-conditions, which can be challenging to define for complex contracts.

With the advent of decentralized finance (DeFi), the scope of vulnerabilities has expanded to include governance attacks, oracle price manipulation, and front-running [80], [81]. Tools like DeFort [20], DeFiRanger [21] and DeFiTainter [22] address DeFi-specific vulnerabilities using transaction analysis and inter-contract taint tracking. Despite these advancements, existing tools often rely on predefined security patterns, limiting their ability to capture sophisticated or emergent vulnerabilities. Recently, LLM-based security analysis tools like GPTScan [16] and PropertyGPT [25] have been proposed to leverage the in-context learning of LLM in the field of static analysis and formal verification for smart contracts. PartitionGPT complements the existing security efforts through performing secure program partitioning to preserving confidentiality for smart contracts.

B. Secure Program Partitioning

Secure program partitioning has been introduced since 2001 by Zdancewic et al. [60] for protecting confidential data during computation in distributed systems containing mutually untrusted host. They developed a program splitter that accept Java program and its security annotations on data variables, and assign according statements to the given hosts.

The program partitioning works can be broadly categorized based on the granularity of code splitting. Function-level program partitioning takes the entire functions as units for separation, and it has been extensively studied [50], [61], [62], [82], [83], [84], [85], [86], [87]. Brumley and Song [50] accept a few annotations on variables and functions, and then partitions the input source into two programs: the monitor and the slave. They apply inter-procedural static analysis, i.e., taint analysis and C-to-C translation. The workflow of our approach is in general similar to them. But, PartitionGPT harnesses LLM's in-context learning for Solidity-to-Solidity translation, and we propose a dedicated equivalence checker to formally verify the correctness of the resulting program partitions. Subsequently, Liu et al. [61] proposes a set of techniques for supporting general pointer in the automatic program partitioning by employing a parameter-tree approach for representing data of pointer types that exist in languages like C. To balance the security and performance and select appropriate partitions, Wu et al. [62] define a quantitative measure of the security and performance of privilege separation. Particularly, they measure the security by the size of code executed in unprivileged process, where the smaller the privileged part is, the more secure the program is. Our work also follows the insight about the security measure during the partition ranking but PartitionGPT uses the edit distance compared to the original program code rather than runtime overhead to select the appropriate partition that could compete with the human-written. There are also program partitioning techniques specialized for creating and deploying partitioning to TEE-based secure environments [53] [82], [83], [84], [85]. For instance, Glamdring [53] uses static program slicing and backward slicing to isolate security-related functions involving sensitive operations in C/C++ programs for execution within Intel SGX enclaves. Different from the abovementioned approach, PartitionGPT creates program partitions for Solidity smart contracts, and deploys partitions to a virtualized TEE-based secure environment tailored for blockchain context that could support different TEE devices.

To achieve finer-grained security, tools like Civet [88] extend partitioning to the statement level. Civet [88], designed for Java, combines dynamic taint tracking and type-checking to partition Java applications while securing sensitive operations inside enclaves. Additionally, Program-mandering (PM) [52] introduces a quantitative approach by modeling privilege separation as an integer programming problem. PM optimizes partitioning boundaries to balance security and performance trade-offs based on user-defined constraints. By quantifying information flow between sensitive and non-sensitive domains, PM helps developers refine partitions iteratively, but it still relies on significant user input, such as defining budgets and goals. Other statement-level partitioning tools include Jif/split [89], which leverages security annotations to enforce information flow policies and partitions Java programs into trusted and untrusted components. Similarly, Swift [51] partitions web applications to ensure that security-critical data remains on the trusted server while client-side operations handle non-sensitive data. DataShield [90] takes a similar approach by separating sensitive and non-sensitive data in memory and enforcing logical separation, though it does not physically split code into separate domains. While these tools offer finer-grained partitioning and improved optimization, they often require significant developer input, such as specifying trust relationships, or iteratively refining partitions. In contrast, PartitionGPT leverage LLM's in-context capability to automate the generation and improvement of program partitions, and the resulting program partitions are verified using a dedicated prover.

VIII. Conclusion

In this work, we present PartitionGPT, the first LLM-driven framework for generating *compilable*, and *verified* secure program partitions to defend against manipulation vulnerabilities for smart contracts. Our evaluation on 18 annotated confidential smart contracts demonstrates that PartitionGPT is able to partition smart contracts with a precision of 80%, reducing around 26% code compared to function-level partitioning. Furthermore, the evaluation results

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on May 15,2026 at 01:46:10 UTC from IEEE Xplore. Restrictions apply.

indicate that PARTITIONGPT could effectively defend against eight of nine real-world manipulation attacks through secure program partitioning, and the runtime overhead is moderate with gas increase between 61% and 101% when deploying partitions into a TEE-based infrastructure. In the future, we plan to enhance PARTITIONGPT by incorporating automated identification of sensitive data variables within smart contracts, enhancing partition ranking technique with external expert models, and further improving the effectiveness of secure program partitioning.

REFERENCES

[1] “Solidity,” 2018. [Online]. Available: https://solidity.readthedocs.io/en/v0.5.1/

[2] G. Wood, “Ethereum: A secure decentralised generalised transaction ledger,” *Ethereum Project Yellow Paper*, vol. 151, pp. 1–32, 2014.

[3] “Binance Smart Chain (BSC),” 2020. [Online]. Available: https://www.bnbchain.org/zh-CN

[4] “Solana: Web3 infrastructure for everyone,” 2020. [Online]. Available: https://solana.com

[5] “Vulnerability: Integer overflow and underflow,” 2023. [Online]. Available: https://owasp.org/www-project-smart-contract-top-10/2023/en/src/SC02-integer-overflow-underflow.html

[6] “Vulnerability: Reentrancy,” 2023. [Online]. Available: https://owasp.org/www-project-smart-contract-top-10/2023/en/src/SC01-reentrancy-attacks.html

[7] “Vulnerability: Front-running attacks,” 2023. [Online]. Available: https://owasp.org/www-project-smart-contract-top-10/2023/en/src/SC05-front-running-attacks.html

[8] “Bad randomness in solidity smart contracts.” Accessed Jan. 20, 2025. [Online]. Available: https://www.immunebytes.com/blog/bad-randomness-in-solidity-smart-contracts/

[9] “What are price oracle manipulation attacks in DeFi?” Accessed Jan. 20, 2025. [Online]. Available: https://www.halborn.com/blog/post/what-are-price-oracle-manipulation-attacks-in-DeFi

[10] “Decoding Jimbo’s protocol exploit.” Accessed Jan. 20, 2025. [Online]. Available: https://quillaudits.medium.com/decoding-jimbos-protocol-7-5m-exploit-quillaudits-772ad1db6c07

[11] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, “Making smart contracts smarter,” in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, New York, NY, USA: ACM, 2016, pp. 254–269.

[12] “Sofware Reliability Lab,” Securify, 2019. [Online]. Available https://securify.ch/

[13] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, “SmartCheck: Static analysis of Ethereum smart contracts,” in *Proc. 1st Int. Workshop Emerg. Trends Softw. Eng. Blockchain,* 2018, pp. 9–16.

[14] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis, “Ethainter: A smart contract security analyzer for composite vulnerabilities,” in *Proc. 41st ACM SIGPLAN Conf. Program. Lang. Des. Implementation,* 2020, pp. 454–469.

[15] Y. Feng, E. Torlak, and R. Bodik, “Precise attack synthesis for smart contracts,” 2019, *arXiv:1902.06067*.

[16] Y. Sun et al., “GPTScan: Detecting logic vulnerabilities in smart contracts by combining GPT with program analysis,” in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng.,* 2024, pp. 1–13.

[17] B. Jiang, Y. Liu, and W. Chan, “ContractFuzzer: Fuzzing smart contracts for vulnerability detection,” in *Proc. 33rd ACM/IEEE Int. Conf. Automat. Softw. Eng.*, New York, NY, USA: ACM, 2018, pp. 259–269.

[18] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, “sFuzz: An efficient adaptive fuzzer for solidity smart contracts,” in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.,* 2020, pp. 778–788.

[19] “Trail of bits.” Echidna. Accessed Jan. 20, 2025. [Online]. Available https://github.com/trailofbits/echidna

[20] M. Xie et al., “DeFort: Automatic detection and analysis of price manipulation attacks in DeFi applications,” in *Proc. 33rd ACM SIGSOFT Int. Symp. Softw. Testing Anal.,* 2024, pp. 402–414.

[21] S. Wu et al., “DeFiRanger: Detecting DeFi price manipulation attacks,” *IEEE Trans. Dependable Secure Comput.*, vol. 21, no. 4, pp. 4147–4161, Jul./Aug. 2024.

[22] Q. Kong, J. Chen, Y. Wang, Z. Jiang, and Z. Zheng, “DeFiTainter: Detecting price manipulation vulnerabilities in DeFi protocols,” in *Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Testing Anal.,* 2023, pp. 1144–1156.

[23] “Symbolic execution tool for smart contracts.” Manticore. Accessed Jan. 20, 2025. [Online]. Available: https://github.com/trailofbits/manticore

[24] “A security analysis tool for EVM bytecode.” Mythril. Accessed Jan. 20, 2025. [Online]. Available: https://github.com/ConsenSys/mythril

[25] Y. Liu et al., “PropertyGPT: LLM-driven formal verification of smart contracts through retrieval-augmented property generation,” in *Proc. 32nd Annu. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, Feb. 2025.

[26] “Securing Web3 with decentralized intelligence.” Certora. Accessed Jan. 20, 2025. [Online]. Available: https://www.certora.com/

[27] E. Hildenbrandt et al., “KEVM: A complete formal semantics of the Ethereum virtual machine,” in *Proc. IEEE 31st Comput. Secur. Found. Symp. (CSF)*, Piscataway, NJ, USA: IEEE Press, 2018, pp. 204–217.

[28] D. Magazzeni, P. McBurney, and W. Nash, “Validation and verification of smart contracts: A research agenda,” *Computer*, vol. 50, no. 9, pp. 50–57, Sep. 2017.

[29] M. Rodler, W. Li, G. O. Karame, and L. Davi, “Sereum: Protecting existing smart contracts against re-entrancy attacks,” 2018, *arXiv:1812.05934*.

[30] A. Li, J. A. Choi, and F. Long, “Securing smart contract with runtime validation,” in *Proc. 41st ACM SIGPLAN Conf. Program. Lang. Des. Implementation,* 2020, pp. 438–453.

[31] Z. Chen, Y. Liu, S. M. Beillahi, Y. Li, and F. Long, “Demystifying invariant effectiveness for securing smart contracts,” *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, pp. 1772–1795, 2024.

[32] X. L. Yu, O. Al-Bataineh, D. Lo, and A. Roychoudhury, “Smart contract repair,” *ACM Trans. Softw. Eng. Method. (TOSEM)*, vol. 29, no. 4, pp. 1–32, 2020.

[33] S. So and H. Oh, “SmartFix: Fixing vulnerable smart contracts by accelerating generate-and-verify repair using statistical models,” in *Proc. 31st ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.,* 2023, pp. 185–197.

[34] T. D. Nguyen, L. H. Pham, and J. Sun, “SGuard: Towards fixing vulnerable smart contracts automatically,” in *Proc. IEEE Symp. Secur. Privacy (SP)*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 1215–1229.

[35] C. Gao, W. Yang, J. Ye, Y. Xue, and J. Sun, “SGuard+: Machine learning guided rule-based automated vulnerability repair on smart contracts,” *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 5, pp. 1–55, 2024.

[36] L. Zhang et al., “ACFix: Guiding LLMs with mined common RBAC practices for context-aware repair of access control vulnerabilities in smart contracts,” *IEEE Trans. Softw. Eng.*, vol. 51, no. 9, pp. 2512–2532, Sep. 2025.

[37] P. Tolmach, Y. Li, and S.-W. Lin, “Property-based automated repair of DeFi protocols,” in *Proc. 37th IEEE/ACM Int. Conf. Automat. Softw. Eng.,* 2022, pp. 1–5.

[38] Y. Zhang, S. Ma, J. Li, K. Li, S. Nepal, and D. Gu, “SmartShield: Automatic smart contract protection made easy,” in *Proc. IEEE 27th Int. Conf. Softw. Anal., Evolution Reengineering (SANER)*, Piscataway, NJ, USA: IEEE Press, 2020, pp. 23–34.

[39] H. Jin, Z. Wang, M. Wen, W. Dai, Y. Zhu, and D. Zou, “AROC: An automatic repair framework for on-chain smart contracts,” *IEEE Trans. Softw. Eng.*, vol. 48, no. 11, pp. 4611–4629, Nov. 2021.

[40] R. Cheng et al., “Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts,” in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS&P)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 185–200.

[41] M. Russinovich et al., “CCF: A framework for building confidential verifiable replicated services,” Microsoft, Redmond, WA, USA, Tech. Rep. MSR-TR-2019-16, 2019.

[42] Y. Yan et al., “Confidentiality support over financial grade consortium blockchain,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data,* 2020, pp. 2227–2240.

[43] Y. Xiao, N. Zhang, J. Li, W. Lou, and Y. T. Hou, “PrivacyGuard: Enforcing private data usage control with blockchain and attested off-chain contract execution,” in *Proc. 25th Eur. Symp. Res. Comput. Secur. (Comput. Secur.–ESORICS)*, Guildford, U.K.: Springer, 2020, pp. 610–629.

[44] R. Yuan, Y.-B. Xia, H.-B. Chen, B.-Y. Zang, and J. Xie, “ShadowETH: Private smart contract on public blockchain,” *J. Comput. Sci. Technol.*, vol. 33, pp. 542–556, May 2018.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on May 15,2026 at 01:46:10 UTC from IEEE Xplore. Restrictions apply.

[45] H. Yin, S. Zhou, and J. Jiang, “Phala network: A confidential smart contract network based on Polkadot,” Phala Network, Singapore, Tech. Rep., 2019. [Online]. Available: https://files.phala.network/phala-paper.pdf
[46] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, “Hawk: The blockchain model of cryptography and privacy-preserving smart contracts,” in *Proc. IEEE Symp. Secur. Privacy (SP)*, Piscataway, NJ, USA: IEEE Press, 2016, pp. 839–858.
[47] B. Bünz, S. Agrawal, M. Zamani, and D. Boneh, “Zether: Towards privacy in a smart contract world,” in *Proc. Int. Conf. Financial Cryptogr. Data Secur.*, Springer, 2020, pp. 423–443.
[48] J. H. Saltzer and M. D. Schroeder, “The protection of information in computer systems,” *Proc. IEEE*, vol. 63, no. 9, pp. 1278–1308, Sep. 1975.
[49] “Secret network: Putting privacy first with decentralized confidential computing,” 2020. [Online]. Available: https://scrt.network/
[50] D. Brumley and D. Song, “PrivTrans: Automatically partitioning programs for privilege separation,” in *Proc. 13th USENIX Secur. Symp. (USENIX Security 04)*, San Diego, CA, USA: USENIX Assoc., Aug. 2004, pp. 57–72.
[51] S. Chong et al., “Secure Web applications via automatic partitioning,” *ACM SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 31–44, 2007.
[52] S. Liu et al., “Program-Mandering: Quantitative privilege separation,” in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, New York, NY, USA: ACM, 2019, pp. 1023–1040.
[53] J. Lind et al., “GlamDring: Automatic application partitioning for intel SGX,” in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, Berkeley, CA, USA: USENIX Assoc., 2017, pp. 285–298.
[54] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” Oct. 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf
[55] G. Almashaqbeh and R. Solomon, “SoK: Privacy-preserving computing in the blockchain era,” in *Proc. IEEE 7th Eur. Symp. Secur. Privacy (EuroS&P)*, Piscataway, NJ, USA: IEEE Press, 2022, pp. 124–139.
[56] R. Solomon, R. Weber, and G. Almashaqbeh, “SmartFHE: Privacy-preserving smart contracts from fully homomorphic encryption,” in *Proc. IEEE 8th Eur. Symp. Secur. Privacy (EuroS&P)*, Piscataway, NJ, USA: IEEE Press, 2023, pp. 309–331.
[57] Q. Ren et al., “Cloak: Transitioning states on legacy blockchains using secure and publicly verifiable off-chain multi-party computation,” in *Proc. 38th Annu. Comput. Secur. Appl. Conf.,* 2022, pp. 117–131.
[58] S. J. Wang, K. Pei, and J. Yang, “SmartInv: Multimodal learning for smart contract invariant inference,” in *Proc. IEEE Symp. Secur. Privacy (SP)*, Piscataway, NJ, USA: IEEE Press, 2024, pp. 2217–2235.
[59] “The liquidity and game theory of new self-marketmaking tokenomics models on Trader Joe v2.1.” Medium. Accessed Jan. 20, 2025. [Online]. Available: https://medium.com/@ultraxbt/the-liquidity-and-game-theory-of-new-self-marketmaking-tokenomics-models-on-trader-joe-v2-1-5357fbaff5b7
[60] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers, “Secure program partitioning,” *ACM Trans. Comput. Syst. (TOCS)*, vol. 20, no. 3, pp. 283–328, 2002.
[61] S. Liu, G. Tan, and T. Jaeger, “PTRSplit: Supporting general pointers in automatic program partitioning,” in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.,* 2017, pp. 2359–2371.
[62] Y. Wu, J. Sun, Y. Liu, and J. S. Dong, “Automatically partition software into least privilege components using dynamic data dependency analysis,” in *Proc. 28th IEEE/ACM Int. Conf. Automat. Softw. Eng. (ASE),* 2013, pp. 11–15.
[63] A. Shirafuji, Y. Oda, J. Suzuki, M. Morishita, and Y. Watanobe, “Refactoring programs using large language models with few-shot examples,” in *Proc. 30th Asia-Pacific Softw. Eng. Conf. (APSEC)*, Piscataway, NJ, USA: IEEE Press, 2023, pp. 151–160.
[64] D. Grubisic, C. Cummins, V. Seeker, and H. Leather, “Compiler generated feedback for large language models,” 2024, *arXiv:2403.14714*.
[65] J. Jiao, S. Kan, S.-W. Lin, D. Sanan, Y. Liu, and J. Sun, “Semantic understanding of smart contracts: Executable operational semantics of solidity,” in *Proc. IEEE Symp. Secur. Privacy (SP)*, Piscataway, NJ, USA: IEEE Press, 2020, pp. 1695–1712.
[66] S.-W. Lin, P. Tolmach, Y. Liu, and Y. Li, “SolSee: A source-level symbolic execution engine for solidity,” in *Proc. 30th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.,* 2022, pp. 1687–1691.
[67] L. De Moura and N. Bjørner, “Z3: An efficient SMT solver,” in *Proc. Int. Conf. Tools Alg. Construct. Anal. Syst.*, Berlin, Heidelberg: Springer, Mar. 2008, pp. 337–340.
[68] “The solidity source analyzer.” Slither. Accessed Jan. 20, 2025. [Online]. Available: https://github.com/crytic/slither
[69] “Confidential EVM smart contracts using fully homomorphic encryption.” Zama. Accessed Jan. 20, 2025. [Online]. Available: https://github.com/zama-ai/fhevm
[70] H. Nir, Y. Avishay, L. Meital, and L. Yair, “COTI V2: Confidential computing Ethereum layer 2,” 2024. [Online]. Available: https://coti.io/files/coti_v2_whitepaper.pdf
[71] “fhEVM.” Zama. Accessed Jan. 20, 2025. [Online]. Available: https://github.com/zama-ai/fhevm
[72] “COTI V2 confidentiality preserving L2: SDKs and examples.” COTI. [Online]. Available: https://github.com/coti-io/confidentiality-contracts
[73] “DeFi hacks reproduce–Foundry.” DeFiHackLabs. Accessed Jan. 20, 2025. [Online]. Available https://github.com/SunWeb3Sec/DeFiHackLabs
[74] A. Sabelfeld and D. Sands, “Declassification: Dimensions and principles,” *J. Comput. Secur.*, vol. 17, no. 5, pp. 517–548, 2009.
[75] “EIP-20: A standard interface for tokens,” Ethereum, 2015. [Online]. Available: https://eips.ethereum.org/EIPS/eip-20
[76] N. Atzei, M. Bartoletti, and T. Cimoli, “A survey of attacks on Ethereum smart contracts (SoK),” in *Proc. 6th Int. Conf., Princ. Secur. Trust (POST), Eur. Joint Conf. Theory Pract. Softw. (ETAPS)*, Uppsala, Sweden: Springer, 2017, pp. 164–186.
[77] “Ganache: A tool for creating a local blockchain for fast Ethereum development.” Accessed Jan. 20, 2025. [Online]. Available: https://github.com/trufflesuite/ganache
[78] J. Feist, G. Grieco, and A. Groce, “Slither: A static analysis framework for smart contracts,” in *Proc. IEEE/ACM 2nd Int. Workshop Emerg. Trends Softw. Eng. Blockchain (WETSEB)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 8–15.
[79] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, “Securify: Practical security analysis of smart contracts,” in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.,* New York, NY, USA: ACM, 2018, pp. 67–82.
[80] L. Zhou et al., “Sok: Decentralized finance (DeFi) attacks,” in *Proc. IEEE Symp. Secur. Privacy (SP)*, Piscataway, NJ, USA: IEEE Press, 2023, pp. 2444–2461.
[81] S. Werner, D. Perez, L. Gudgeon, A. Klages-Mundt, D. Harz, and W. Knottenbelt, “Sok: Decentralized finance (DeFi),” in *Proc. 4th ACM Conf. Adv. Financial Technol.,* 2022, pp. 30–46.
[82] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia, “Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation,” in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.,* 2015, pp. 1607–1619.
[83] A. Ghosn, J. R. Larus, and E. Bugnion, “Secured routines: Language-based construction of trusted execution environments,” in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC),* 2019, pp. 571–586.
[84] N. Santos, H. Raj, S. Saroiu, and A. Wolman, “Using ARM TrustZone to build a trusted language runtime for mobile applications,” in *Proc. 19th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, Mar. 2014, pp. 67–80.
[85] K. Rubinov, L. Rosculete, T. Mitra, and A. Roychoudhury, “Automated partitioning of android applications for trusted execution environments,” in *Proc. 38th Int. Conf. Softw. Eng.,* 2016, pp. 923–934.
[86] K. Gudka et al., “Clean application compartmentalization with SOAAP,” in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.,* 2015, pp. 1016–1031.
[87] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, “Enforcing Kernel security invariants with data flow integrity,” in *Proc. 23rd Annu. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, San Diego, CA, USA, Feb. 2016.
[88] C.-C. Tsai, J. Son, B. Jain, J. McAvey, R. A. Popa, and D. E. Porter, “Civet: An efficient Java partitioning framework for hardware enclaves,” in *Proc. 29th USENIX Secur. Symp. (USENIX Secur.)*, Berkeley, CA, USA: USENIX Assoc., 2020, pp. 505–522.
[89] L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic, “Using replication and partitioning to build secure distributed systems,” in *Proc. Symp. Secur. Privacy*, Piscataway, NJ, USA: IEEE Press, 2003, pp. 236–250.
[90] S. A. Carr and M. Payer, “Datashield: Configurable data confidentiality and integrity,” in *Proc. ACM Asia Conf. Comput. Commun. Secur.,* 2017, pp. 193–204.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on May 15,2026 at 01:46:10 UTC from IEEE Xplore. Restrictions apply.