Towards Automatic Discovery of Denial of Service Weaknesses in Blockchain Resource Models

Feng Luo
The Hong Kong Polytechnic University
Hong Kong, China
University of Electronic Science and Technology of China
Chengdu, China

Huangkun Lin
University of Electronic Science and Technology of China
Chengdu, China

Zihao Li
The Hong Kong Polytechnic University
Hong Kong, China

Xiapu Luo*
The Hong Kong Polytechnic University
Hong Kong, China

Ruijie Luo
University of Electronic Science and Technology of China
Chengdu, China

Zheyuan He
University of Electronic Science and Technology of China
Chengdu, China

Shuwei Song
University of Electronic Science and Technology of China
Chengdu, China

Ting Chen*
University of Electronic Science and Technology of China
Chengdu, China

Wenxuan Luo
University of Electronic Science and Technology of China
Chengdu, China

Abstract

Denial-of-Service (DoS) attacks at the execution layer represent one of the most severe threats to blockchain systems, compromising availability by depleting the resources of victims. To counteract these attacks, many blockchains have implemented unique resource models that incorporate transaction fees. Nevertheless, historical incidents of DoS attacks demonstrate that these resource model designs remain inadequate. Although there are studies that manually craft DoS attacks on specific blockchains in isolation, none of them can discover DoS weaknesses in blockchains automatically. In this paper, we provide an insight into DoS weaknesses in blockchain resource models, and present a generic and systematic approach to uncover these weaknesses. In our approach, we first identify DoS weaknesses by DoSVER, a novel tool that reasons feasible DoS weaknesses against blockchain resource models by formal verification. The identified DoS weaknesses will be further validated by DoSDET, a new framework that automates the attack synthesis in exploiting the identified DoS weaknesses. We conduct a comprehensive and systematic evaluation by extensive experiments on nine diverse and widely-used blockchains, and discovered 12 DoS weaknesses with corresponding exploitation across the nine blockchains, 10 of which were unveiled for the first time.

CCS Concepts

• **Security and privacy → Distributed systems security**.

*Corresponding author

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690329

Keywords

Blockchain, Denial-of-Service Weaknesses

ACM Reference Format:
Feng Luo, Huangkun Lin, Zihao Li, Xiapu Luo, Ruijie Luo, Zheyuan He, Shuwei Song, Ting Chen, and Wenxuan Luo. 2024. Towards Automatic Discovery of Denial of Service Weaknesses in Blockchain Resource Models. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3658644.3690329

1 Introduction

Backed by contracts as logic, various industries (e.g., traditional finance and online games) [47] have been reshaped by blockchains. Currently, blockchains supporting contracts have swept the globe in recent years with a total market capitalization of 5,49B USD [9].

Economically valuable cryptocurrencies make blockchains attractive targets for malicious attackers. For example, a notorious DoS attack caused a 2-3x performance reduction in Ethereum nodes' execution [2] by forcing these nodes into busy I/O with executing EXTCODESIZE repeatedly. The DoS attack seriously affected the availability of Ethereum and shook the market price of its native cryptocurrency [1]. Even worse, since each node needs to replay all historical transactions to maintain a complete copy of the blockchain state, this kind of execution-layer DoS attacks also slow down the execution performance of all newly enrolled nodes [25].

To counter execution-layer DoS attacks, various blockchains employ resource models to charge users transaction fees via refined charging mechanisms to prevent unlimited consumption of blockchain resources. However, with unsound resource models, blockchains remain vulnerable to DoS attacks that consume physical or on-chain execution resources [44, 45]. Unfortunately, no existing works provide an automatic approach to discover these DoS weaknesses in blockchains due to three limitations. **L1:** Most existing research [30, 35, 48] cannot identify the underlying causes of

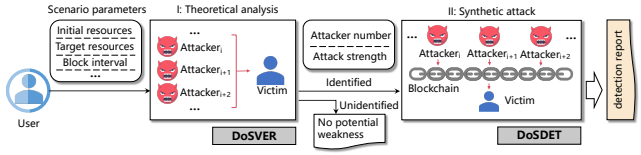


Figure 1: The workflow of our approach. After determining the scenario parameters, DoSVER explores the scenarios against various attacker numbers and attack strengths to identify if there are DoS weaknesses under the current resource model. DoSDET further constructs attack scenarios based on the output of DoSVER and synthesizes attacks to exploit the identified weaknesses. Finally, DoSDET reports DoS weaknesses if the corresponding attacks are valid.

execution-layer DoS attacks, i.e., weaknesses in the resource model of blockchains [42, 54]. **L2:** Existing works only studied Ethereum or EOS [26, 27, 32], ignoring numerous other blockchains with different architectures. **L3:** Previous studies discover DoS weaknesses by manual analysis [44, 54]. The differences between blockchains make understanding new mechanisms and usages time-consuming. The diversity in blockchain source code and contract languages requires users to be familiar with specific languages for manual code examination and contract writing, making the manual analysis process cumbersome and difficult to extend to other blockchains.

In this paper, we focus on the execution-layer DoS problem and conduct the first systematic approach for discovering platform-agnostic DoS weakness. Our approach can automatically discover four resource-related DoS weaknesses by two phases (cf. Fig.1). In Phase I, we design DoSVER (§4), a formal verification tool with blockchain resource models to reason DoS weaknesses. By specifying the scenarios with input parameters (e.g., initial resources, target consumed resources, blockchain generation interval), DoSVER can reason whether DoS weaknesses exist, and output attack scenarios (e.g., attacker numbers and attack strength) that exploit the weakness in seconds or minutes. In Phase II, we present DoSDET (§5), an automated attack synthesis framework, for validating the identified DoS weaknesses. Specifically, DoSDET constructs attack scenarios based on the output of DoSVER, synthesizes attacks exploiting DoS weaknesses, and measures corresponding metrics to evaluate whether the corresponding DoS attacks are valid.

However, it is non-trivial to achieving such a systematic approach due to three challenges. **C1:** It is challenging to implement a generalized formal verification tool with complicated and diverse resource models. **C2:** To automatically synthesize attacks, we need to construct contracts with specific languages, and launch attacks by manipulating clients with specific control commands. However, the corresponding information is difficult to obtain automatically from diverse implementation of blockchains. **C3:** The metrics used for evaluating DoS attacks rely on runtime information, which cannot be directly accessible. To capture these metrics, we need to automatically instrument blockchains. However, the diversity of blockchain implementation makes this task difficult.

To address the above challenges, we propose new techniques. Specifically, to address **C1**, we introduce a transaction resource theory that describes transaction operation and resource consumption rules. Besides, we design a general model that specifies the behavior of blockchain users, and defines the verification method for DoS problems (§4). For addressing **C2**, we design an interactive process that utilizes large language models (LLM) to analyze and capture all needed information (§5.2). To address **C3**, we design a general-purpose parser that supports various languages (e.g., Java, Go, C++, and Rust) based on lexical analysis and syntax analysis. The parser can locate the functions that process blocks and interpret for all VM instructions, and insert probe to obtain runtime information with corresponding language (§5.3).

Table 1: DoS weaknesses in nine blockchains

Weakness	Ethereum	Tron	RSK	Klaytn	EOS	WAX	BOSCore	Telos	Solana
SRD	✗	✗	✗	✗	✓	✓	✓	✓	✗
FR	✗	✗	✗	✗	✗	✗	✓	✗	✗
MP	✗	✓	✓	✓	✗	✗	✗	✗	✗
RPCS	✓	✓	✓	✓	✗	✗	✗	✗	✗

SRD (SCP Resource Drain): can be exploited to drain CPU/RAM resources in victim account. FR (Free Resource): can be exploited to drain CPU resources in victim nodes at a lower cost, MP (Mispricing): can be exploited to cause delays for victim nodes, RPCS (RPC Service): can be exploited to slow down the block synchronization rate (✗: weakness do not exist, ✓: weakness exists, ✓: weakness exists and was first discovered by us).

We apply our approach to *nine* representative blockchains with different architectures. Columns 2-4 of Table 2 show their technical differences in terms of program language, contract language, and the interval in generating new blocks. With extensive experimental results, we obtained insightful findings (cf. Table 1). Based on Common Weakness Enumeration (CWE) [8], we dissected the design issues of blockchain resource models that cause these weaknesses (§7). At the time of writing, our findings have been reported to the corresponding blockchain development teams.

In summary, we make the following contributions.

- We conduct the first systematic study for automatically uncovering DoS weaknesses on diverse blockchains.
- We present the first formal verification tool for identifying DoS weaknesses at the blockchain system level.
- We develop DoSDET, the first generalized attack synthesis framework that automatically validate DoS weaknesses by synthesizing corresponding attacks against the weaknesses.
- We conduct comprehensive evaluation for our approach on nine distinct and representative blockchains. With extensive experimental results, we reveal various DoS weaknesses, demonstrating the effectiveness of our approach. We release our materials of our work in https://github.com/sec-study-dev/dos-tool.

2 Background

This section provides background related to blockchain, resource model, and threat model.

2.1 Blockchain

Smart contract. Smart contracts are programs that define a set of rules for governing associated funds, typically written in a Turing-complete program language (e.g., Solidity) [47, 59]. After being compiled into bytecode, smart contracts will be deployed to the blockchain and executed by blockchain virtual machine (VM) according to the predefined logic [61, 65, 68].

Blockchain virtual machine. A blockchain VM is a virtual state machine that formally specifies blockchain's execution model (i.e., how the system state is altered given a series of bytecode and a small tuple of environmental data) [37, 62]. Different blockchains can

Table 2: The technical differences of the nine blockchains

Blockchain	Program language	Contract language	Block interval	Resource model				
				QR	Process mode	Fee bearer	Free quota	Fee calculation basis
Ethereum	Go	Solidity	12s	Gas	M1	Caller	✗	Instruction price
Tron	Java	Solidity	3s	Bandwidth	M2,M4	Caller	5000 bandwidth	
				Energy	M2	Shared	✗	
RSK	Java	Solidity	3s	Gas	M1	Caller	✗	
Klaytn	Go	Solidity	3s	Gas	M1	Caller	✗	
EOS	C++	C++	0.5s	CPU	M2	Caller	✗	Resource consumption
WAX				NET	M2	Caller	✗	
Telos				RAM	M3	Caller	✗	
BOSCore	C++	C++	0.5s	CPU	M2, M4	Caller	200000 ms	
				NET	M2, M4	Caller	10 KB	
				RAM	M3	Caller	✗	
Solana	Rust	Rust	0.4s	SOL	M1	One signer & owner	✗	Number of signatures

Note that in EOS, WAX, Telos and BOSCore there are two types of QR named CPU and RAM, which do not refer to the actual physical resources of CPU and RAM.

customize their VMs as needed. Appendix A provides the technical differences between the various blockchain VMs.

2.2 Resource Model

Resource type. There are two types of resources in blockchain, the first is the resources of the physical machine running the blockchain node (e.g., CPU, memory, IO), denoted as NR. The second is the custom quantifiable resources (e.g., Gas in Ethereum) introduced by the blockchain developers to characterize the consumption of NR during transaction execution and used as a unit of transaction fee, denoted as QR.

Resource model. Since smart contracts are executed by VM embedded in the blockchain client, this process consumes NR of each node. To prevent NR misuse, the blockchain introduces transaction fee, where any given fragment of programmable computation is executed with the corresponding QR deducted from the transaction fee bearer [66]. Specifically, current mainstream blockchains have three transaction fee calculation rules: (i) Pricing custom instructions by accruing the cost of the instruction being executed. (ii) Conversion based on the consumption of NR by the executed fragment. (iii) Introducing multi-signature transactions and calculating transaction fees based on the number of signatories.

For the acquisition and processing of QR, there are four types: **M1:** Transactions directly consume the fee bearer's native currency based on a specified ratio of QR to native currency. **M2:** Accounts get QR based on market prices by staking native currency, which can be redeemed at any time and has a fixed redemption cycle. The transaction consumes the QR of the fee bearer, and the consumed part is automatically recovered within a certain period. **M3:** Accounts purchase QR at market prices by using native currency. Transactions will consume QR owned by the transaction fee bearer, and such resources will not be automatically recovered after consumption. **M4:** For blockchains with a free resource mechanism, each account will be given a fixed amount of free QR, which will be preferentially consumed during transactions.

Due to the difference in design, the transaction fee bearer in a blockchain is one of the following four situations: (i) Caller of the smart contract (i.e., Caller). (ii) Owner of smart contracts (i.e., Owner). (iii) Owner can set the proportion (0% - 100%) of the transaction fee it undertakes, and the rest is paid by the caller (i.e., Shared) [5]. (iv) The first signer of a multi-signature transaction. Differences in resource management mechanisms among the nine blockchains are shown in columns 5-9 of Table 2, presented in terms of QR names, resource processing modes, fee bearer, free resource quotas, and rules for calculating the transaction fee.

2.3 Threat Model

Our threat model is depicted in Fig. 2. The attacking node can possess multiple blockchain accounts, which can be used to sign attack transactions to the victim. Depending on the type of resource targeted by the attack, the attacker can exhaust the QR of the victim account or the NR of the victim node to impair the normal service of the victim at a low cost. The victim will then disseminate transactions and blocks to the rest of the blockchain network.

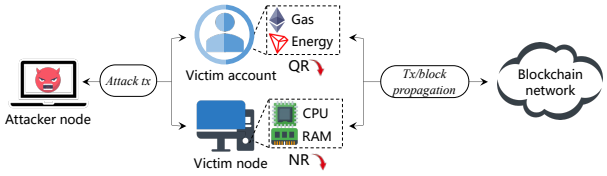


Figure 2: Threat model of our work

3 Resource-related DoS weaknesses

This section describes the four DoS weaknesses related to the resource model and analyzes their prerequisites. Our research focuses on discovering DoS weaknesses tied to the native resource model. Therefore, other DoS attacks (e.g., network layer and consensus layer) are out of the scope of our work.

SCP Resource Drain Weakness (SRD). In blockchains represented by EOS, a contract is allowed to generate deferred transactions to execute another contract via defer action (cf. Fig. 9, line 5). Contracts can consume the contract owner's QR (e.g., RAM, CPU) to generate delayed transactions, but require the owner to grant `eosio.code` permissions to it. Contract owners often grant to use their resources for deferred transactions to facilitate the broad adoption of their contracts. Attackers can exploit this weakness to dry off the owner's QR and cause its contracts to be unavailable [42]. Depending on the target consumed, attacks exploiting SRD can be classified as SCP CPU drain (CPUD) and RAM drain (RAMD) attacks.

Free Resource Weakness (FR). To attract more users to the blockchain ecosystem, some blockchains have proposed a free resource mechanism. This allows users to pay for transactions in priority when executing them through a fixed amount of free resources gifted by the system. While this mechanism makes normal

users convenient, it also creates opportunities for malicious attackers. We first discovered this weakness. Although the designers limit the number of free resources to ensure security, they ignore the power of free resources in combination with other attack methods. The attackers can exploit this free resource mechanism to reduce attack costs even to no cost and consume the victim's QR.

Mispricing Weakness (MP). Blockchains represented by Ethereum calculate transaction fees through VM instructions contained in executed contracts [54]. Each instruction has a specific cost and is expressed in its units (e.g., gas, energy), which are set by the designers of the blockchain VM. However, through experiments, Perez et al. have proven that the cost of instructions can potentially be set too low compared to resource usage [54], meaning that an attacker can invoke a contract containing massive amounts of under-priced opcodes to waste many NR at low cost [24].

RPC Services Weakness (RPCS). The blockchain remote procedure call (RPC) service emerges as an intermediary that bridges the two to enable distributed applications to interact with the blockchain. However, the speculative smart contract execution interface in the RPC service introduces a DoS vulnerability. The interface is considered speculative because it can execute any smart contract for free (e.g., `eth_call()` in Ethereum, `TriggerSmartContract()` in Tron). Although such executions will only run locally on the serving RPC node and will not propagate to other nodes, recent work has shown that attacks targeting a single node can exhaust the NR of a victim node with severe consequences. [44].

Weakness Prerequisites. We can observe that the above weaknesses are all related to some specific aspect of the blockchain mechanisms, which are prerequisites for the weaknesses to exist. We summarize these prerequisites as the basis for automated weakness discovery by DoSDET (§5.2) and set our framework to detect only those weaknesses with the prerequisites to reduce overhead (§5.4). Specifically, for SRD, it requires the blockchain to have defer action and `eosio.code` privilege ($\mathcal{P}_{SRD}$). For FR, it requires the blockchain to have a free resource mechanism ($\mathcal{P}_{FR}$). For MP, it requires the blockchain to price opcodes and calculate transaction fees based on all opcodes executed by the contract ($\mathcal{P}_{MP}$). For RPCS, it requires blockchain to support RPC services that include speculative execution interface ($\mathcal{P}_{RPCS}$).

4 DoSVER

This section presents our formal tool for blockchain DoS attacks. Unlike one-time program vulnerability exploits, successful DoS attacks should cause prolonged service unavailability in real-time systems, not just tolerable short interruptions [22]. Ignoring the effect of time when assessing a real-time system's susceptibility to DoS attacks may lead to inaccurate results, and a robust verification model should consider the time factor [22]. Formalizing a real-time system is challenging due to the need to model physical time progression and multiple concurrent processes [23]. To address this, Ölveczky et al. propose defining real-time systems in rewriting logic as real-time rewriting theory, and demonstrate the benefits of this approach [53]. Rewriting logic models communication and concurrency abstractly, allowing customization of execution strategies within the logic itself and transforming verification goals into reachability problems. Compared to traditional formal methods like Kronos [29] and UPPAAL [28], rewriting logic is better suited to the execution system. Given an abstract property the system should satisfy, it attempts to verify that situations exist in the system execution that meet the given property. Kanovich et al. further propose timed multiset rewriting (MSR), a language framework for real-time rewrite logic, showing that symbolically solving reachability problems based on it is PSPACE-complete [40]. Thus, we employ timed MSR in constructing DoSVER.

4.1 Overview

To build a generic model, instead of modeling each blockchain or resource model separately, we implement a parameterized model by abstracting the inherent behavior of the blockchain system itself. By inputting different initial parameters, it can systematically verify whether DoS attacks can succeed in scenarios with different blockchain resource mechanisms and attack types. Specifically, we model two aspects of the blockchain: transaction execution and user behavior. ① For transactions, we define transaction specifications that model the execution process and resource consumption, with each transaction altering the blockchain state (§4.3). ② For users, we propose user models to describe the behavior of deploying and invoking contracts through transactions. A user's role can be assigned as either an attacker or a victim (§4.4). The transaction model and user model are compatible with various blockchains and cover various resource models (§2.2) by specific input parameters.

In DoSVER, each user can initiate transactions independently. Transactions execute in an environment with multiple users (both attackers and victims), forming a labeled transformation system via rewriting rules to simulate multi-transaction concurrency in the blockchain. The DoS problem in blockchain is defined as a trace reachability problem, and corresponding formal validation goals are established based on different DoS attack objectives (§4.5). As shown in Fig. 1, DoSVER accepts scenario parameter inputs to instantiate a parameterized model into various blockchain mechanisms and attacks. These include initial resource amounts, target resources consumed by the attack (NR or QR), attack type, block interval time, and resource recovery time (if automatic resource recovery is present in the blockchain). Then, DoSVER creates attack scenarios with any number of attackers, who can launch attacks of varying strengths. For an attack scenario, DoSVER solves the trace reachability problem. If there is a trace that satisfies the validation goal, it indicates the DoS attack can effectively impact the blockchain system in this scenario. DoSVER incrementally increases the number of attackers and adjusts their attack strengths to validate the impact of different attack levels on the blockchain. For scenarios where valid attacks are verified, DoSVER outputs the number of attackers and attack strengths corresponding to the scenarios, guiding DoSDET to generate the corresponding actual attacks (§5.5).

4.2 Model Related Definitions

This section introduces the relevant concepts involved in our formalization. In DoSVER, the state of the blockchain system encompasses users' internal states, node resources, transactions' internal state, and messages. The system state, described as a configuration, is represented by a set of timestamp facts. The timestamp facts are of the form $F@T$, where F is a fact taking a fixed number of terms as

parameters, and T is a non-negative real number called a timestamp. A special predicate symbol $Time$ with zero arity (i.e., no parameters) indicates the global time and appears only once in a configuration. Transaction execution and user actions are specified as multiset rewrite rules, which define the transitions between configurations.

Rewrite rules. The rules are divided into advancement and instantaneous rules, depending on their impact on the system. ① Advancement rules advance the global time in the form of $Time@T \rightarrow Time@(T+\varepsilon)$, where ε is a positive value. ② Instantaneous rules modify facts in the configuration other than global time in the form of $L \xrightarrow{C} R$, where L/R is a set of facts representing premises/conclusions, respectively, and C is a set of time constraints representing action guards. An instantaneous rule applies to a given ground state configuration $\mathcal{S}$ if there is a ground substitution σ such that $L\sigma \subseteq \mathcal{S}$ and $C\sigma$ is true. The resulting configuration $\mathcal{S}' = (\mathcal{S}/L) \cup R\sigma$. The instance of the applied rule r rewrites the configuration $\mathcal{S}$ as $\mathcal{S}'$ noted as the one-step relation $\mathcal{S} \rightarrow_r \mathcal{S}'$.

Trace. For a given set of rules $\mathcal{R}$, we specify that the trace is a sequence of configurations starting from $\mathcal{S}_0$ i.e., $\mathcal{S}_0 \rightarrow_{r_1} \mathcal{S}_1 \rightarrow \cdots_{r_n} \mathcal{S}_n$ such that $0 \leq i \leq n-1$, $r_i \in \mathcal{R}$. In our rewrite logic, rewrite rules are time-robust [56], namely: i) In any given state, an advancement rule can advance time by an arbitrary amount up to a specific time instant. ii) An instantaneous rule can be applied only if there is time advancement in the system. Thus, the interleaving of all traces can cover the entire time-following state space of the system. More intuitively, by specifying the set of traces, the transformation system simulates all possible sequences of events [39].

Therefore, developing a suitable goal configuration and performing a trace search can verify whether certain events we are concerned about will occur. The verification goal is specified in a fragment of the first-order logic represented as a trace. However, the number of traces covering all cases is immense, and it is laborious and pointless to focus on all possible traces. Besides, since rules can be applied automatically in an uncertain way in transition system, there can be traces indicating transactional misbehavior. To reduce the search space, we additionally introduce uncritical configuration. The uncritical configuration is used to exclude all the traces that are not relevant to our validation goal. We provide the definitions of uncritical and goal configurations below.

Definition 4.1 (Uncritical/Goal Configurations.). An uncritical (resp. goal) configuration is specified by an uncritical configuration specification $\mathcal{US}$ (resp. goal $\mathcal{GS}$) which is a set of pairs $\langle \mathcal{F}_1, C_1 \rangle, \ldots, \langle \mathcal{F}_n, C_n \rangle$, where $\mathcal{F}_i$ is a set of timestamp facts and C_i is the time constraint associated with the facts. A configuration $\mathcal{S}$ is an uncritical configuration w.r.t. $\mathcal{US}$ (resp. a goal configuration w.r.t. $\mathcal{GS}$) if for $1 \leq i \leq n$, there is a grounding substitution, σ, such that $\mathcal{F}_i\sigma \in \mathcal{S}$ and $C_i\sigma$ evaluates to true.

We define a trace as critical when its decomposition by global time progression yields all sub-traces without uncritical configurations. For example, for a uncritical configuration specification $\langle \{Time@T, F@T_1\}, \{T_1 = T\} \rangle$, the trace $[Time@1,\ F@2] \longrightarrow_{r_1} [Time@3, F@2]$ will be considered critical because it does not contain any uncritical configurations. But trace $[Time@1,\ F@2] \longrightarrow_{r_2} [Time@2, F@2] \longrightarrow_{r_3} [Time@3, F@2]$ will not be considered critical because it contains uncritical configurations $\{Time@2 = F@2\}$. By customizing the uncritical configurations that apply to blockchain transactions (rf. §4.3), DoSVER can focus only on critical traces to avoid the interference of irrelevant traces and reduce search overhead. Next, we detail the first-order type alphabet needed to elaborate our formalism.

Resource. We define NR and QR as arrays R and r, respectively, consisting of natural number items whose number of items is specific to different blockchain resource models. Each transaction tx is associated with an NR consumption R_{tx} and a QR consumption r_{tx}, representing the resources required to execute tx. Considering the shared cost bearing introduced in §2.2 and the fact that a blockchain satisfying $\mathcal{P}_{SRD}$ can consume contract owner resources through transactions, we consider an r_{tx} as consisting of r_c and r_o, which represent the number of resources consumed by the contract caller and the contract owner, respectively, during the transaction. In addition, the initial resources of the blockchain node and the user are denoted as R_{ini} and r_{ini}, respectively.

Predicates. The global time predicate $Time$. The fact $\mathsf{S}_i(tx, R, c_{id})$, where $0 \leq i \leq n$, specifies the $n+1$ transaction states for transaction tx. R denotes the NR that needs to be allocated during the maintenance of normal execution of tx. The c_{id} denotes the contract associated with this transaction, and if tx is a transaction that creates a contract, c_{id} is recorded as $\oslash$ before the contract is deployed, and the timestamp t of S_i denotes that tx is in state S_i until moment t. The fact $\mathsf{User}(id, r)@t$ denotes that user id / the initiator of transaction id has r resources available at time t is available. $\mathsf{Owner}(c_{id}, r)@t$ denotes that the owner of the contract c_{id} has r resources available at time t. $\mathsf{Node}(R)@t$ denotes that the blockchain node has R node resources available at time t. The fact $\mathsf{N(m)}@t$ denotes that the message m is available for reception in the transaction from moment t. The fact $\mathsf{M}(id, m)@t$ denotes that the user id knows the message m at moment t. $\mathsf{Av}(tx)@t$, $\mathsf{Bl}(tx)@t$ and $\mathsf{Rej}(tx)@t$ represent the transaction tx is available or blocked or rejected from from moment t, respectively. $\mathsf{Rec}(id, t)@t$ denotes that the attacker id can recover r resources at moment t.

4.3 Transaction Specifications

We first build a generic model for transactions that is capable of describing how transactions are processed in blockchains with different resource models. To this end, we extract knowledge from various transaction management mechanisms and define a transaction resource theory that formalizes the transaction process and resource consumption. To the best of our knowledge, it applies to all current mainstream blockchains.

Definition 4.2 (Transaction $\mathcal{RT}$). For a tx, its transaction resource theory (transaction $\mathcal{RT}$) is specified by three state predicates S_0, S_1, S_2, and three rules, i.e., transaction execution, transaction terminated and transaction availability rules.

In the definition, S_0: the state upon initiation of the contract creation transaction, S_1: either the state upon completion of the contract creation or upon completion of the transaction calling the contract, and S_2: the state upon initiation of the transaction calling the contract. The transaction termination rule establishes that a transaction ceases to execute when the transaction fee bearer cannot cover the fee. The transaction availability rules indicate that transactions are blocked when node resources are inadequate and become available once resources are sufficient. We show the

$$\textbf{CRE:}\begin{bmatrix} Time@T \\ \mathsf{S}_0(tx, R_{tx}, \oslash)@T_1 \\ \mathsf{User}(tx, r + r_{tx})@T_2 \\ \mathsf{Node}(R + R_{tx})@T_3 \\ \mathsf{N}(CRE)@T_4 \end{bmatrix} \xrightarrow[T_2,T_3,T_4 \le T]{T_1 \ge T} \exists c_{id}. \begin{bmatrix} Time@T \\ \mathsf{S}_1(tx, R_{tx}, c_{id})@(T+\delta) \\ \mathsf{User}(tx, r)@T \\ \mathsf{Node}(R)@T \end{bmatrix}$$

$$\textbf{INV:}\begin{bmatrix} Time@T \\ \mathsf{S}_1(tx, R_{tx}, c_{id})@T_1 \\ \mathsf{User}(tx, r)@T_2 \\ \mathsf{Node}(R)@T_3 \\ \mathsf{N}(INV)@T_4 \end{bmatrix} \xrightarrow[T_2,T_3,T_4 \le T]{T_1 \ge T} \begin{bmatrix} Time@T \\ \mathsf{S}_2(tx, R_{tx}, c_{id})@(T+\delta) \\ \mathsf{User}(tx, r)@T \\ \mathsf{Node}(R)@T \end{bmatrix}$$

$$\textbf{EXE:}\begin{bmatrix} Time@T \\ \mathsf{S}_2(tx, R_{tx}, c_{id})@T_1 \\ \mathsf{User}(tx, r + r_c)@T_2 \\ \mathsf{Owner}(tx, r + r_o)@T_3 \\ \mathsf{Node}(R)@T_4 \\ \mathsf{N}(EXE)@T_5 \end{bmatrix} \xrightarrow[T_2,T_3,T_4,T_5 \le T]{T_1 \ge T} \begin{bmatrix} Time@T \\ \mathsf{S}_1(tx, R_{tx}, c_{id})@(T+\delta) \\ \mathsf{User}(tx, r)@T \\ \mathsf{Owner}(tx, r)@T \\ \mathsf{Node}(R + R_{tx})@T \end{bmatrix}$$

Figure 3: Transaction Execution Rules

transaction execution rules in Fig. 3 and the other two rules and considerations of operational semantics are detailed in Appendix B.1.

Rules **CRE**, **INV**, and **EXE** specify the transitions between the states as described above. **CRE** reflects the process by which users deploy new contracts. **INV** reflects that the user initiates a transaction to invoke the contract. **EXE** reflects the completion of the transaction of invoking the contract. Appendix B.2 provides concretized examples of transaction execution rules to aid understanding.

As stated in §4.2, the interleaving of all traces generated by the rewrite rule can cover any situation, which guarantees that applying our transaction $\mathcal{RT}$ can characterize the correct execution of any transaction tx in any situation as expected. However, in addition to the expected execution, there will be extraneous redundant traces, and an uncertain application of the rules may generate traces representing transaction misbehavior. To refuse the waste of validation search capabilities for such traces, we define the following transaction uncritical configuration specification applicable to blockchain systems based on a basic definition of uncritical configurations (Definition 4.1) to focus our attention on critical traces.

Definition 4.3 (Transaction $\mathcal{US}$). The transaction uncritical configuration specification (transaction $\mathcal{US}$) for a given transaction theory of transaction tx, consisting of four types of uncritical configuration specifications:

- **Terminated $\mathcal{US}$**: for all $0 \le i \le n$

  $\langle \{Time@T, \mathsf{S}_i(tx, R_{tx}, c_{id})@T_1, \mathsf{User}(tx, r_c - r)@T_2\}, \{T_1 > T, T_2 < T\}\rangle$;

  $\langle \{Time@T, \mathsf{S}_i(tx, R_{tx}, c_{id})@T_1, \mathsf{Owner}(tx, r_o - r)@T_2\}, \{T_1 > T, T_2 < T\}\rangle$;
- **Rejected $\mathcal{US}$**:

  $\langle \{\mathsf{Node}(R_{tx})@T_1, \mathsf{Av}(tx)@T_2\}, \{T_1 < T_2\}\rangle$;
- **Available $\mathcal{US}$**:

  $\langle \{\mathsf{Node}(R_{tx})@T_1, \mathsf{Bl}(tx)@T_2\}, \{T_1 < T_2\}\rangle$;
- **Creatable $\mathcal{US}$**:

  $\langle \{\mathsf{User}(tx, r_c)@T_1, \mathsf{Owner}(tx, r_o)@T_2, \mathsf{Rej}(tx)@T_3\}, \{T_1 < T_3, T_2 < T_3\}\rangle$;

Terminated $\mathcal{US}$ specifies that the configuration that represents a terminated transaction is uncritical. Rejected $\mathcal{US}$ specifies that configurations are uncritical if a transaction is considered available when its resources have been exhausted. Available $\mathcal{US}$ dictates that transactions should not be blocked at any time when NR is sufficient. Creatable $\mathcal{US}$ specifies that transactions must be successfully created whenever QR is sufficient. Appendix B.3 demonstrates the impact of the absence of the above transaction $\mathcal{US}$. By applying transaction $\mathcal{US}$, DoSVER can exclude all irrelevant or misbehavioral traces and only need to focus on critical traces. Next, based on transaction $\mathcal{RT}$ and transaction $\mathcal{US}$, we provide a formalism for transactions in the blockchain.

Definition 4.4 (Transaction). Formally, a transaction is a triplet $\mathcal{T} = (tx, \mathcal{RT}, \mathcal{US})$ where:

- tx is a unique identification symbol;
- $\mathcal{RT}$ is a set of transaction $\mathcal{RT}$ involving only tx;
- $\mathcal{US}$ is a set of transaction $\mathcal{US}$ corresponding to $\mathcal{RT}$.

The sets of transaction state predicates of different transaction theories are disjoint. Therefore, multiple transactions can be initiated at the same time in our model to satisfy the highly concurrent nature of blockchain systems. When quantifying NR, storage resources are quantified as positive numbers, while computational resources are represented as a percentage of available CPU. Each operation consumes resources for a given time.

4.4 Specifying User Model

To model users, we first normalize user behavior. Intuitively, a user can freely create or invoke contracts as long as his QR is sufficient. For generality, we introduce a consideration of the QR recovery mechanism and formalize the user behavior rules in Fig. 4. Each rule has an associated cost specified by the COST function, which returns a triplet $\langle \delta_T, \delta_t, \delta_r \rangle$, where δ_T is the time to execute the transaction; δ_r represents the number of QR consumed by the transaction, which becomes available again after δ_t units of time.

Specifically, a user advances a transaction by receiving and sending a message reflecting each decomposition rule of the transaction execution rules. The **DEP**/**CALL** rules describe the action of contract creation/call by users. Transactions initiated by the above user actions are processed according to the corresponding transaction execution rules in the transaction $\mathcal{RT}$ and are constrained by the transaction termination rule and the transaction availability rule (cf. §4.3). **RES** rule represents the recovery of consumed resources. This rule only applies to blockchains with a resource recovery mechanism, i.e., where the scenario parameters input to the DoSVER include a resource recovery time. We then define blockchain users.

Definition 4.5 (User). A blockchain user $\mathcal{U}$, consists of a unique identification symbol U, an initial resource r^U_{ini}, a finite set $\mathcal{M}$ =

Action Rules:

$$\textbf{DEP:}\begin{bmatrix} Time@T \\ \mathsf{M}(U, CRE)@T_1 \\ \mathsf{User}(U, r + \delta_r)@T_2 \end{bmatrix} \xrightarrow{T \ge T_1} \begin{bmatrix} Time@T \\ \mathsf{M}(U, INV)@(T + \delta_T) \\ \mathsf{User}(U, r)@T \\ \mathsf{Rec}(U, r)@(T + \delta_t) \end{bmatrix}$$

where $\mathrm{COST}_{\mathrm{DEP}}(U, CRE) = \langle \delta_T, \delta_t, \delta_r \rangle$

$$\textbf{CALL:}\begin{bmatrix} Time@T \\ \mathsf{M}(U, INV)@T_1 \\ \mathsf{User}(U, r + \delta_r)@T_2 \end{bmatrix} \xrightarrow{T \ge T_1} \begin{bmatrix} Time@T \\ \mathsf{M}(U, EXE)@(T + \delta_T) \\ \mathsf{M}(U, INV)@T \\ \mathsf{User}(U, r)@T \\ \mathsf{Rec}(U, \delta_r)@(T + \delta_t) \end{bmatrix}$$

where $\mathrm{COST}_{\mathrm{CALL}}(U, INV) = \langle \delta_T, \delta_t, \delta_r \rangle$

Resource Recovery Rules:

$$\textbf{RES:}\begin{bmatrix} Time@T \\ \mathsf{User}(U, r)@T_1 \\ \mathsf{Rec}(U, \delta_r)@T_2 \end{bmatrix} \xrightarrow{T \ge T_2} \begin{bmatrix} Time@T \\ \mathsf{User}(U, r + \delta_r)@T \end{bmatrix}$$

Figure 4: User Specification Rules

$\{M(U, m_1)@0, \ldots, M(U, m_n)@0\}$ of M facts specifying user's initial knowledge base and definitions of each rule given in Fig.4.

Users in different blockchains are specified by setting different COST_R functions. For example, $\text{COST}_\text{R}(C, U) = \langle \delta_T, [r_1, r_2], [t_1, \infty] \rangle$ means that executing transactions in a blockchain consumes two QRs, r_1 and r_2, in which the consumed r_1 will be recovered after t_1, while r_2 will not be recovered. Furthermore, by specifying role attributes for the user model, we can create any number of attackers $\mathcal{A}_1, \ldots, \mathcal{A}_n$ and victims $\mathcal{W}_1, \ldots, \mathcal{V}_n$ in DoSVER.

4.5 DoS Problem

To verify that the DoS attack is successful, we first define the blockchain DoS problem. The purpose of a DoS attack in the blockchain is to exhaust the target's NR or QR, thereby rendering the blockchain system incapable of serving normal users. Moreover, a valid DoS attack should cause prolonged service unavailability rather than short interruptions that can be tolerated [22]. We formulate DoS attacks in our model based on these concepts. When the node's NR or victim's QR is exhausted for some duration (Δ), the DoS attack on the blockchain is successful.

Specifically, a victim can be determined to be subject to a valid DoS attack initiated by attackers by searching for critical traces of the form: $\mathcal{S}_0 \to \mathcal{S}_1 \to \cdots \to \mathcal{S}_i \to \cdots \to \mathcal{S}_{i+m} \to \cdots \to \mathcal{S}_n$, where $\mathcal{S}_0$ is the initial configuration of the verification scenario, the global time t_i of configuration $\mathcal{S}_i$ and the global time t_{i+m} of configuration $\mathcal{S}_{i+m}$ such that $t_{i+m} - t_i \geq \Delta$, and that for victim initiated transaction tx, its NR or QR in all configurations between $\mathcal{S}_i$ and $\mathcal{S}_{i+m}$ are less than R_{tx} or r_{tx}. To facilitate the illustration of our model, we set the verification scenario to contain only one victim.

Definition 4.6 (Verification Scenario). Formally, a verification scenario in blockchain system is a seven-tuple $\mathcal{V} = (\Phi, \Psi, V, \Delta, \mathcal{IS}_\mathcal{V}, \mathcal{GS}_\mathcal{V}, \mathcal{US}_\mathcal{V})$ where:

- Φ is a finite set of transactions $\mathcal{T}_1, \ldots, \mathcal{T}_n$ (see §4.3);
- Ψ is a finite set of attacker $\mathcal{A}_1, \ldots, \mathcal{A}_n$;
- V is a victim user;
- $\Delta \in \mathbb{N}$ specifies the minimal duration which represent a successful DoS attack;
- $\mathcal{IS}_\mathcal{V}$ is the initial configuration of $\mathcal{V}$;
- $\mathcal{GS}_\mathcal{V}$ is the goal configuration of $\mathcal{V}$;
- $\mathcal{US}_\mathcal{V}$ is a union consisting of all uncritical configuration specifications of all transactions $\mathcal{T}_i$.

In this definition, $\mathcal{IS}_\mathcal{V}$ containing exactly the following timed facts, for $1 \leq i \leq m$: (1) $Time@0$, specifying that the initial global time is 0; (2) $\text{User}(A_i, r_{ini}^{A_i})@0$ represents the initial state of the attacker $\mathcal{A}_i$; (3) $R(V, r_{ini}^{V})@0$ represents the initial state of the victim V; (4) $\text{Node}(R_{ini})@0$, where R_{ini} is the maximal resource of the blockchain node; (5)$\mathcal{M}_i$, the initial knowledge base of $\mathcal{A}_i$. In addition to, due to the different types of resources that the four DoS attacks we focus on consume victims, $\mathcal{GS}_\mathcal{V}$ comes in two forms. More precisely, for attacks targeting the victim QR (SRD and FR):

$$\mathcal{GS}_\mathcal{V} = \{\langle \{Time@T, \text{Rej}(tx_i)@T_1\}, \{T \geq T_1 + \Delta\} \rangle\},$$

and for attacks targeting the victim NR (MP and RPCS):

$$\mathcal{GS}_\mathcal{V} = \{\langle \{Time@T, \text{Bl}(tx_i)@T_1\}, \{T \geq T_1 + \Delta\} \rangle\}.$$

where $1 \leq i \leq n$, and tx_i is the unique identification symbol of some $\mathcal{T}_1$ related to $\mathcal{V}$.

Lemma 4.1. Blockchain DoS problem is an instance of the critical trace reachability problem in the verification scenario.

We provide the proof of Lemma 4.1 in Appendix C. This lemma indicates that the blockchain DoS problem can be successfully transformed into a scenario-specific trace reachability problem, and solving such problems based on rewrite logic is PSPACE-complete [40]. Therefore, we can define the blockchain DoS problem as follows.

Definition 4.7 (Blockchain DoS problem). Let $\mathcal{V}$ be a verification scenario. The blockchain DoS problem is to determine whether there is a critical trace w.r.t. $\mathcal{US}_\mathcal{V}$ from $\mathcal{S}_\mathcal{V}$ to a goal configuration w.r.t. $\mathcal{GS}_\mathcal{V}$.

By simply instantiating $\mathcal{V}$, DoSVER can reason about various resource-related DoS problems under different attack scenarios. We set a validation time of up to 10 minutes for DoSVER, and when a timeout or failure occurs, DoSVER automatically increases the number of attackers and attack strength to detect stronger attack scenarios.

Example. We use an example to demonstrate the verification of RPCS in Ethereum. Since the RPCS attack does not consume QR, all parameters related to QR are specified as 0. Ether has no resource recovery mechanism, so there is no need to enter the relevant parameters. In the verification scenario, we set the initial resource of the blockchain node R_{ini}=100, which is the target resource consumed by the RPCS attack. Attackers knows the information from the transaction resource theory, i.e., the set of timed facts:

$$\mathcal{M} = \{M(A, CRE)@0.0, M(A, INV)@0.0, M(A, EXE)@0.0\}.$$

Thus, the initial configuration is:

$$\mathcal{M} \cup \{\, Time@0, \text{User}(A, 0)@0, \text{Node}(100)@0, \text{Av}(tx)@0\}.$$

The goal configuration is shown in §4.5. We set the DoS attack is successful when transactions continue to be unavailable for 5 blocks. Therefore, by inputting the block interval of Ethereum (cf. Table 2), the goal configuration for RPCS in Ethereum is:

$$\mathcal{GS}_\mathcal{V} = \{\langle \{Time@T, \text{Bl}(tx_i)@T_1\}, \{T \geq T_1 + 60\} \rangle\}.$$

After determining the initial and goal configurations based on the input parameters, DoSVER creates an attack scenario with multiple attackers and a victim. The attackers simulate transactions to create the attack contract using the **DEP** action and to invoke the attack contract using the **CALL** action continuously (§4.4). DoSVER adjusts the number of attackers and the strength of their attacks to verify multiple scenarios. Transactions are processed based on the transaction $\mathcal{RT}$. After applying all transaction $\mathcal{US}$ to exclude irrelevant traces, DoSVER searches for traces that satisfy the goal configuration among the remaining critical traces (§4.3). If such a trace exists, it indicates that the DoS attack effectively impacts the system in this scenario. DoSVER then outputs the corresponding number of attackers and attack strength.

5 DoSDET

5.1 Overview

To realize a generic automated attack synthesis process, DoSDET includes four key components (cf. Fig. 5). Step ①: Info Collector gathers information on blockchain control commands and weakness prerequisites through a question-answer (QA) interaction with ChatGPT [11]. Step ②: Probe Inserter inserts probes into the blockchain VM with $\mathcal{P}_{MP}$, which records essential runtime information, and

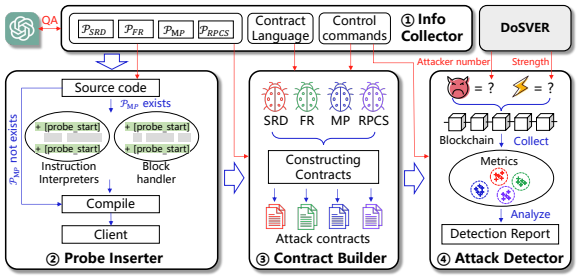


Figure 5: Overview of DoSDET

compiles the source code with completed or without probe insertion to build the blockchain client. Step ③: Contract Builder generates attack contracts that exploit weaknesses. Step ④: Attack Detector constructs an attack scenario based on the output of DoSVER and launches an attack using the generated contract. It verifies the existence of weaknesses by measuring the impact of the attack and finally outputs a threat report.

5.2 Step ①: Info Collector

To automatically generate contracts to exploit weaknesses and launch attacks, DoSDET needs to know the contract language of target blockchains and manipulate the client using control commands. Besides, each weakness has a corresponding prerequisite, and meaningless overhead can be avoided if specific prerequisites can be identified from the blockchain design ahead of time, and attack synthesis can be performed only for the weaknesses corresponding to them (§3). Therefore, as a precondition for implementing an attack synthesis framework, we need to obtain answers to 3 questions: ❶ What is the contract language supported by the target blockchain? ❷ Does the target blockchain satisfy some attack prerequisites? ❸ How can we extract various blockchain control commands and transform them into a form to be exploited in real time?

However, answering the above questions is challenging because their answers are distributed across a variety of mediums, including blockchain documents and websites. These mediums have different styles, making manual analysis inefficient and error-prone. Especially when working on a new blockchain, even professional researchers may encounter problems and spend a lot of time solving them. Inspired by the ability of LLM to parse these human-readable contents efficiently, we design promotion strategies for QA interactions with ChatGPT (currently the best overall performing LLM) to achieve efficient automated information extraction.

Due to its strong advanced reasoning capabilities, ChatGPT is very accurate in answering simple deterministic questions such as ❶ and ❷. However, ChatGPT's output has an inherent randomness when faced with complex questions like ❸, and thus, relying entirely on its output will have a high error rate. This problem can usually be overcome by pre-training, but this approach only improves ChatGPT's accuracy when parsing problems with high correlation to the train dataset, which is not in line with our aim of supporting various blockchains. Therefore, we design a problem decomposition method to decompose ❸ into two simpler deterministic problems of command parameter type determination and parameter matching.

Specifically, for ❶ and ❷, we directly ask ChatGPT to answer them. For ❸, we first ask ChatGPT to find out all the parameter types of a specific command for the target blockchain, and then ask

```
System: You are a blockchain expert. You will be asked questions
        related to blockchain mechanisms. You can mimic the
       answers to these questions five times in the background
       and provide me with the most frequent answers. In addition
       , please strictly adhere to the output format specified in
        the questions and do not explain your answers.
Question ❸
Please list the parameter types of [%Command Type%] for
      [%Blockchain Type%] and organize the results into an array
      format.
Answer: [%Parameter Types%]
Find an available [%Command Type%] and replace the corresponding
      value in the command with the appropriate type based on
      the parameter type in [%Parameter Types%] and output the
      result as a template.
```

Figure 6: Prompt for question ❸

it to find out a usable command example, ensure the correctness of the command and its format, and obtain the command template by matching and replacing the parameter types with the corresponding variables in the example. The subsequent process of DoSDET can use this command by changing the parameter type in the command template to real-time values. Fig. 6 shows the prompts for ❸, the prompts for ❶ and ❷, and all the information Info Collector can gather as shown in Appendix D. To maximize accuracy, we adopt the "mimic-in-the-background" prompt approach [60]. We minimize the error rate by instructing the model to mimic answering the question five times in the background and providing the most frequent answers. In this way, the Info Collector enables a generalized information extraction process without pre-training. In addition, considering that for some less-used blockchains, LLM may be inaccurate due to insufficient learning samples, we additionally add a manual parameter input option to Info Collector to enhance the reliability of DoSDET applied to less-used blockchains.

5.3 Step ②: Probe Inserter

The metric for evaluating MP requires recording the block execution time (§5.5). To construct contracts for MP on-demand(§5.4), DoSDET needs to collect the ratio of CPU, Memory, IO resources to execution cost (i.e., t/c ratio, m/c ratio, io/c ratio) for each instruction. Hence, Probe Inserter needs to insert probes into block handler and all VM instruction interpreters to collect the information. To do this, Probe Inserter first converts different source codes into a generic Token stream suitable by lexical analysis, and then constructs an Abstract Syntax Tree (AST) based on predefined syntax rules. Finally, Probe Inserter locates the target function by locating subtrees with block handling and instruction interpretation semantics.

Converting source codes to Token stream. We analyzed various blockchain source codes for function naming and designed a common set of Token by combining the language features of Java, C++, Golang and Rust. We implement our lexical analyzer based on Flex[43], which contains a set of rules, each containing a regular expression pattern and an action. More than the general lexical analysis rules, we pay extra attention to the sequence of characters that may indicate the name of the target function and record the position of all keywords that may indicate the beginning and end of the target segment through the *yylineno* variable in Flex. We present our positioning process as an example of an ADD interpreter in RSK (Fig. 7(a) to (b)). The function doADD is converted to ADD Token. The outermost left/right braces that are checked by `returnLbrace()`/`returnRbrace()` as immediately adjacent to ADD are

converted to LB/RB with positional information. Other characters are converted according to general rules.

Converting Token stream to AST. To mine semantic information from the Token stream, we implement a parser that generates ASTs using BNF [43] custom grammar rules. For blockchains developed in Java, C++, Golang, and Rust, we identify a fixed pattern for their block handlers and VM instruction interpreters, with the former being an independent function and the latter in the form of either an independent function or a switch-case structure. This result provides the basis for us to obtain the target subtree structure. Taking the ADD interpreter in RSK as an example. The parser shown in Fig. 7(c) reduces the Token stream as an AST, and then Probe Inserter locates the ADD interpreter by finding the subtree shown in Fig. 7(d), where assignment statement `assign`, equation statement `expr`, etc. are reduced according to universal rules. When the target subtree is found, Probe Inserter can find the location of the target function via the *yylineno* variable and insert a probe to capture runtime information via the probe insertion algorithm in Appendix E. Probe Inserter is currently applicable to most mainstream blockchains, as most of them are written in the four languages mentioned above.

5.4 Step ③: Contract Builder

Contract Builder can construct attack contracts for DoS weaknesses. Contract Builder currently supports Solidity, C++ and Rust. To ensure the practicality of the attack contract, Contract Builder constructs contracts conforming to the appropriate attack patterns for SRD, FR and RPCS based on the template, and constructs contracts that repeat a single instruction for MP based on data dependency. All attack contracts contain an attack function with a *payload* parameter to facilitate setting the number of cycles of the attack action based on the attack strength output by DoSVER (e.g., line 2 of Fig. 11).

Contract generation based on templates. Based on the existing research [42, 44] and our previous experiments, we have concluded the practical attack behavior patterns for SRD, FR and RPCS, and designed corresponding templates.

• SRD: Contracts exploiting SRD (CPUD/RAMD) weaknesses have a victim operation (VA) (cf. Fig. 9/Fig. 21), which generates delayed transactions via deferred actions to consume the CPU/RAM of the smart contract provider [42]. An attacker can carry out an attack by calling VA multiple times.

• FR: A reasonable way to exploit the free resource weakness is to enhance the RAMD attack. So for FR, DoSDET reuses the attack contract for the RAMD weakness (cf. Fig. 21).

• RPCS: An effective way to exploit the RPCS is to consume the node's CPU by calling contracts containing massive hash computations (cf. Fig. 11) through speculative contract execution [44], so DoSDET builds such contracts for RPCS.

Contract generation based on instruction data dependency. According to existing research, an effective way to exploit the MP weakness is to invoke a contract that contains a large number of underpriced instructions [54], so DoSDET constructs inline assembly contracts for UP that repeat a single instruction. For instructions whose return results can be used as their own parameters, we design multiple nesting methods for them, and for other instructions, we design them to execute multiple times in parallel in one loop. Since the operational targets of different instructions are different, the key is to prepare an appropriate runtime environment to meet the data dependencies of different instructions. In addition to instructions that require information from the VM (e.g., block number) to directly interact with the private chain, constructing the contract considers the following three cases:

• If the instruction manipulates the stack/memory/storage, DoSDET adds assignment/MSTORE/SSTORE operations to the attack contract to generate data at the corresponding location. The assignment operations push data to the stack, and MSTORE/SSTORE write data to the specified location of memory/storage, respectively.

• If the instruction requires information related to the smart contract (e.g., contract address), DoSDET generates a simple read/write as an auxiliary contract and uses it as a target.

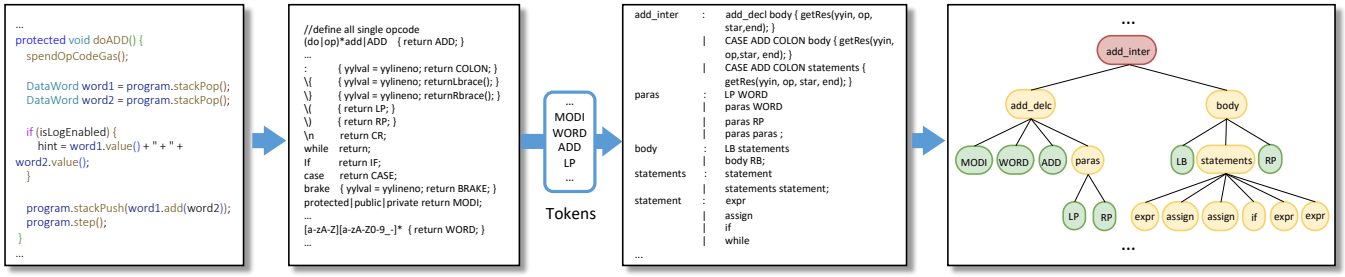
• If the instruction needs to interact with another account, we consider the following three cases. First, if the target account does not exist, no special preparation is required. Second, if the target account is EOA, DoSDET generates one using the account command. Third, if the target account is a smart contract, DoSDET designates it as an auxiliary contract.

On-demand contract generation. Recall that Step ① identifies various weakness prerequisites, so Step ③ only needs to construct contracts for the weaknesses corresponding to the identified prerequisites to reduce overhead. Moreover, intuitively, for attacks against MP weaknesses, the underpricing degree (resource consumption per unit time) of the exploited instructions is positively correlated with the effectiveness of the attack. Therefore, there is no need to build attack contracts for each instruction. For this, Contract Builder synchronizes the mainnet to replay a sufficient number of transactions and collects the three metrics of t/c, m/c, and io/c ratio for each instruction in the real case through the probe. It constructs contracts by comparing these three metrics across all instructions and selecting the two highest instructions in each metric.

5.5 Step ④: Attack Detector

Attack Detector creates the appropriate number of attackers and adjusts the attack strength by modifying the *playload* parameter in the attack contract based on the output of DoSVER. By launching attacks against the corresponding weaknesses and collecting the corresponding evaluation metrics, DoSDET can evaluate the actual impact of each attack on the blockchain and determine the existence of weaknesses. Rather than simply outputting whether a weakness exists, DoSDET output provides details of the attack process for further analysis. Due to the differences in the characteristics of the four weaknesses, we formulate different evaluation metrics for them and detail the simulation attack steps for each weakness in Appendix F.

SRD attack. We adopted the detection metric in [42]. For the CPUD weakness, we use the ratio of CPU consumed by the victim to the attacker during the attack to evaluate the effect of the attack. *If this ratio is greater than 1, it means that the CPU usage of the victim is greater than that of the attacker, which means that a CPUD weakness exists* and the weakness severity is positively related to the ratio. For RAMD weakness, we focus on whether attackers can *exploit recoverable CPU to drain the victim's unrecoverable RAM.*



(a) Interpreter of ADD instruction in RSK (b) Lexer snippet (c) Parser snippet (d) ADD interpreter subtree in AST

Figure 7: Locate ADD instruction interpreter

FR attack. If attackers can *consume a large amount of the victim's non-free resources using only the free CPU given by the system*, we consider that the blockchain has FR.

MP attack. Inspired by [54], we measure the impact of this attack in terms of the execution time of blocks filled with MP attack transactions. Each blockchain node re-executes and verifies a new block when it is received. As shown in Table 2, each blockchain has a certain interval for new block generation. *If the execution time of a block filled with attack transactions is longer than the block generation interval, nodes subjected to such a block will be too busy with validation to synchronize the new block*, resulting in a system slowdown. In this case, we consider that the blockchain exists MP.

RPCS attack. Referring to the criteria in [44], we use the degree of block synchronization slowdown of a local full node during the attack to evaluate the effect of the RPC attack. *If the synchronization rate of the victim node slows down by more than 20%, we consider that the blockchain has* RPCS. In addition, we describe how to circumvent the protection mechanism of the RPC service in the Appendix G.

5.6 Extensibility

In our design, Info Collector performs linguistic analysis tasks by interacting with ChatGPT. Attack Detector performs a fixed number of simulation attack steps, and the steps are specific to the target weakness rather than the blockchain. Thus, the work of these two components is loosely coupled to the specific blockchain design, and no modification of these two components is required when extending to other blockchains.

Probe Inserter takes as input the source code with $\mathcal{P}_{MP}$ and outputs the modified code. Contract Builder takes contract language information and weakness type as input and constructs the corresponding attack contract. Thus, the work of these two components is tightly coupled to the specific blockchain design. When extending to a blockchain where the source code or the smart contract language is new, new rules need to be introduced manually to both components at once. In practice, however, the programming languages we support now cover the vast majority of mainstream blockchains. For the top 20 blockchains with the highest market share [4], the existing framework is able to support 17 of them.

6 Evaluation

We implement DoSVER in Maude [14] by 3,163 lines of codes and DoSDET by 4,213 lines of Python codes. Maude is a widely used tool for rewriting logic [22, 39, 56]. It provides diverse functional modules and supports custom extensions that allow us to encode the semantics of DoSVER's rewrite logic into executable programs. Additionally, Maude provides explicit state analysis methods to observe and verify that the system is executing as expected [52]. After inputting the required parameters of DoSVER, it automatically applies our rewrite rules and output the number of attackers and attack strength of the corresponding attack scenario after successfully verifying a valid DoS attack. Since our rewriting logic is time robust and describes the state at any given moment, the constructed model can fully encompass any possible situation [56]. Checking with this model allows all traces to be searched with enough time, greatly reducing false negatives and positives [39]. However, in the implementation, we impose a 10-minute upper limit for the checking time to ensure the detection efficiency, which may lead to false positives (FP) and false negatives (FN) due to timeout. In the later evaluation, we analyze the FP and FN of DoSVER. Appendix H details the implementation and how *payload* parameter of the attack contract in DoSDET is guided based on the output of DoSVER. Our experiments are driven by four research questions. **RQ1**: How is the performance of DoSVER in analyzing potential DoS weaknesses (§6.2)? **RQ2**: How is the performance of DoSDET in verifying and discovering DoS weaknesses (§6.3)? **RQ3**: Does DoSVER suffer from FP and FN (§6.4)? **RQ4**: What is the overhead of our approach (§6.5)?

6.1 Setup

To fully evaluate the generalizability and effectiveness of our approach, we categorized mainstream blockchains based on the resource model design and chose top-ranked representatives by market capitalization at the time of writing among the various categories [21], a total of nine blockchains: Ethereum, EOS, Tron, RSK, Klaytn, WAX, BOSCore, Telos, and Solana. To meet the hardware configuration requirements of various blockchains [10, 12, 17–19], we experimented on servers equipped with 16-core Intel Xeon Gold 5218R CPU, 2 TB SSD, and 24 GB RAM. To avoid jeopardizing the mainnet and other test blockchain networks, we conduct attack impact assessment experiments on the private chain. The evaluation metrics for each weakness are described in §5.5.

6.2 RQ1: Performance of DoSVER

To reflect the level of impact of an attack in the formal model, we extract the global time forward amount from the searched traces, which is a value that reflects the moment of success of a DoS attack. This means that for an attack scenario, if a global time forward can

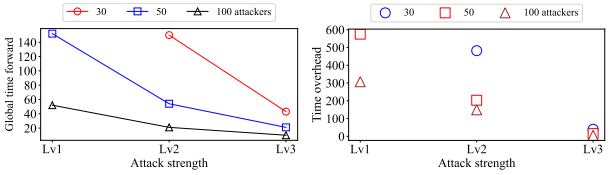
Table 3: Detection results of DoSVER

Blockchain	Attackers	Weakness	Global Time Forward Attack strength Lv1	Lv2	Lv3	Weakness	Global Time Forward Attack strength Lv1	Lv2	Lv3
EOS	30	CPUD	✗	125	74	RAMD	162	106	71
	50		✗	79	42		94	62	39
	100		100	55	28		78	36	13
WAX	30		✗	✗	105		✗	125	79
	50		✗	✗	41		128	92	43
	100		✗	✗	23		93	46	24
Telos	30		✗	153	78		✗	123	68
	50		122	74	39		116	79	38
	100		84	47	20		84	41	17
BOSCore	30		✗	✗	97		✗	✗	71
	50		✗	✗	42		137	87	39
	100		✗	44	19		95	48	22
Ethereum	30	RPCS	✗	✗	✗	MP	✗	✗	✗
	50		✗	✗	✗		✗	✗	✗
	100		152	124	67		✗	✗	✗
Tron	30		✗	✗	✗		✗	✗	✗
	50		✗	✗	136		✗	126	46
	100		141	107	71		169	71	24
RSK	30		✗	✗	✗		✗	✗	✗
	50		✗	✗	✗		✗	92	39
	100		✗	177	95		147	61	22
Klaytn	30		✗	✗	✗		✗	✗	✗
	50		✗	✗	117		✗	✗	46
	100		161	129	62		157	62	27
Solana	30		✗	✗	✗	-	-	-	-
	50		✗	✗	✗		-	-	-
	100		✗	✗	✗		-	-	-
BOSCore	30	FR	184	126	72		-	-	-
	50		104	58	29		-	-	-
	100		61	29	13		-	-	-

✗ indicates no potential weaknesses. ▒ indicates the false negatives of the DoSVER analyzed in §6.4.

be extracted, it indicates the presence of a potential DoS weakness, and vice versa, it indicates that an effective DoS attack cannot be performed in this scenario. For a more detailed evaluation, we use DoSVER to analyze attack scenarios formed by interleaving three different levels of strength and three different numbers of attackers, where the execution times of level 1, level 2, and level 3 attacks are 0.1, 0.3, and 0.5 units of time, respectively.

Table 3 shows all the results. We analyze the CPUD weakness in EOS as an example. Fig. 8a and Fig. 8b demonstrate the detection results and time overhead. It can be seen that when the number of attackers is 30, the DoS weakness may be effectively exploited only when the attack intensity is greater than level 2. When the number of attackers is large enough (e.g., 50/100), low-intensity attacks may also be effective. Attack effectiveness is negatively correlated with the global time forward of the system. These results can be obtained in seconds to minutes. Note that we do not claim that the global time forward values in the results are exactly the same as the numbers when an actual attack is performed since the actual trade execution may also be affected by other factors (e.g., disk IO). However, the results in §6.4 show that DoSVER is sufficient to accurately verify the presence of DoS weaknesses in the resource model within an acceptable margin of error.



(a) Verification results **(b) Verification time overhead**

Figure 8: EOS CPUD weakness automated verification

Table 4: Weakness precondition detection

Weakness	Ethereum	Tron	RSK	Klaytn	EOS	WAX	BOSCore	Telos	Solana
$\mathcal{P}_{SRD}$	◐	◐	◐	◐	○	○	○	○	◐
$\mathcal{P}_{FR}$	◐	◐	◐	◐	◐	◐	○	◐	◐
$\mathcal{P}_{MP}$	○	○	○	○	◐	◐	◐	◐	◐
$\mathcal{P}_{RPCS}$	○	○	○	○	◐	◐	◐	◐	○

○: prerequisite exists, ◐: no prerequisite exists.

Answer to RQ1: DoSVER *is effective in detecting potential DoS weaknesses in resource models.*

6.3 RQ2: Performance of DoSDET

In Step ①, DoSDET can accurately obtain the contract language, control commands, and weakness prerequisites for each blockchain in an average of 22.3s (cf. Table 10) for subsequent steps, which shows the effectiveness of the prompting strategy we designed. Table 4 shows the weakness prerequisites for each blockchain. DoSDET synthesizes attacks against weaknesses where corresponding preconditions exist to reduce overhead. Based on the potential weaknesses verified by DoSVER, DoSDET constructs attack scenarios consisting of up to three numbers of attackers (30, 50, and 100) and three strengths of attacks for each weakness. Due to space limitation, this section shows the results when the number of attackers is 100, and other results can be found in Appendix I.

• **SRD weakness.** The contract builder constructs contracts for CPUD weaknesses and RAMD weaknesses, respectively. For the CPUD weakness, taking the contract constructed for EOS in Fig. 9 as an example, `cpuconsume()` is an exploitable VA, where `txn.actions.emplace_back()` (line 6) creates a defer action to consume CPU resources. An attacker can trigger this VA by calling the contract and adjust the attack strength by changing the incoming *playload*.

CPUD detection. Based on the strength of the attack output by DoSVER, Attack Detector sets the *payload* parameters to 50, 100, and 1000. DoSDET utilizes these contracts to carry out the attack and measures the effectiveness of the attack using the ratio of the CPU cost ratio of victim and attacker as a metric (cf. §5.5). Table 5 shows the results obtained. It can be seen that the CPU cost ratios of EOS and Telos exceed 1 in all cases, while the CPU cost ratios of WAX and BOSCore are below 1 when the *payload* is small and above 1 when the *payload* is large. Therefore, all four blockchains have CPUD weakness that can be exploited, and the weaknesses of EOS and Telos are easier to exploit. Also, we find that EOS and Telos are more affected when using the same victim-attacker contract. With a high enough *payload*, the attacker can even cost the victim more than 20x the cost of the attack in terms of CPU resources.

```
1  void cpuconsume(name user,uint64_t payload){ // VA
2    require_auth(user);
3    eosio::transaction txn;
4    const uint128_t sender_id = (((uint128_t)(current_time_point
         ().sec_since_epoch()) << 64) | (uint128_t)(user.value))
         ;
5    txn.actions.emplace_back(
6    action(permission_level{get_self(),"active"_n},
7    get_self(),
8    "cpuvuln"_n,
9    std::make_tuple(payload)));
10   txn.delay_sec = 0;
11   txn.send(sender_id, get_self(),false);}
```

Figure 9: Code snippets of contract of CPU drain attack

Table 5: Comparing CPU cost ratio of 4 blockchains.

Blockchain	Victim's CPU			Attacker's CPU			CPU cost ratio		
	payload			payload			payload		
	50	100	1000	50	'100	1000	50	100	1000
EOS	112,848	220,040	1,435,779	61,170	63,517	61,412	1.84	3.46	23.37
Telos	108,350	230,614	1,486,548	63,795	62,872	60,411	1.70	3.67	24.61
WAX	32,532	52,067	372,954	53,808	52,970	53,647	0.60	0.98	6.95
BOSCore	29,463	48,018	365,737	47,924	48,181	50,673	0.61	1.00	7.22

Table 6: RAM drain on victim RAM from one attack

Blockchain	Attacker's CPU (ms)			Victim's RAM (KiB)		
	payload			payload		
	50	100	1000	50	'100	1000
EOS	20.4	21.1	20.2	62.5	125	1,250
Telos	20.6	20.7	21.9	62.5	125	1,250
WAX	21.3	22.8	21.1	62.5	125	1,250
BOSCore	16.2	16.8	17.6	62.5	125	1,250

RAMD detection. Attack Detector sets the *payload* parameters to 5, 10, and 100 and allocates 128MiB of RAM resources to the victim. The attacker initiates the attack using CPU resources, by calling the `ramconsume()` (Appendix J) in the victim contract to consume the victim's RAM resources. Table 6 shows the impact of all attackers launching one attack on the victim. According to the resource model of these four blockchains (Table 2), CPU can recover themselves (M2) and RAM can only be purchased through currency (M3). This means that attackers can consume the victim's non-recoverable resources by consuming recoverable resources. Under continuous experiment, the 128 MiB staked for the victim was exhausted within minutes. Based on the market price on October 16, 2023, 128 MiB is worth $1,158 and $2,942 on EOS and WAX, respectively [15].

• **FR weakness.** As described in §5.4, the attack contract constructed for the FR weakness is the same as the contract shown in Appendix I.1.

Weakness detection. Attack Detector sets the *payload* parameters to 5, 10, and 100. All attackers do not receive additional CPU except for the free CPU allocated by the system at the time of creation. Attack Detector allocates enough RAM to the victim to measure the impact on the victim's RAM when the attacker makes full use of the free resources to launch an attack. As can be seen in Table 7, the attacker successfully consumes the victim's RAM using the free resources and reaches a RAM consumption of 694.3 MiB at a *payload* of 100. Based on the market price on October 16, 2023, the consumed RAM is worth $64.49 [7].

• **MP weakness.** In Step ③, Contract Builder first collects t/c, m/c and io/c ratio of related instructions by synchronizing the main network and analyzes them. The results show that the m/c and io/c ratio of the instructions of the four blockchains are fairly balanced, and there are no cases of improper pricing with exponential differences (§5.4). Due to space limitation, we show the m/c and io/c ratio of these instructions in Appendix I.2. Therefore, DoSDET next synthesize the attack only from the perspective of the t/c ratio of the instructions. Contract Builder compares the t/c ratio of these instructions and selects the two instructions with the highest values.

Table 7: QR consumption of the attackers and the victim

Payload	Attacker's CPU (ms)	Victim's RAM (MiB)
5	19,914.3	35.5
10	19,917.0	71.6
100	19,819.8	694.3

Table 8: Comparison of instructions for the four blockchains

	Instruction	Sample count	Mean time (ns)	t/c ratio
Ethereum	SMOD	136,472	872.5	160.7
	EXP	3,517,728	1,463.7	153.9
Tron	EXTCODESIZE	284,572	33,314	1,665.7
	SLOAD	1,476,591	59,293	1,185.9
RSK	RETURNDATASIZE	210,022	484.7	242.4
	DIV	1,323,894	832.8	166.6
Klaytn	SHR	102	526.5	175.5
	DIV	680,849	837.0	167.4

Table 8 shows the cost, sample count, average execution time and t/c rate of these instructions. As we set, Contract Builder constructs the attack contract based on these instructions.

Contract Generation. EXTCODESIZE will return the code size of an address, so DoSDET directly use the address of the auxiliary contract deployed first. SLOAD will read the data at the x position through `sload(x)`, so in the contract we need to use `sstore(x, y)` to insert the y value at the x position. RETURNDATASIZE returns the size of the last returndata, so no parameters are used. SMOD, EXP, SHR and DIV represent signed modulo remainder operation, exponential operation, right shift operation and division operation respectively, therefore, in the contract, it is necessary to write two data in the stack through the assignment operation first. Besides, they have the feature that the result can be used as the parameter of the next same instruction. Taking the DIV instruction of RSK as an example, Contract Builder will generate a contract as shown in Fig. 10.

Weakness detection. Table 9 shows the results of detecting MP on four blockchains. In Ethereum, after using the SMOD and EXP instructions to perform attacks, the execution times of the blocks filled with attack transactions are 10.36s and 10.19s, respectively. This is lower than the block generation interval of Ethereum, and the victim nodes subjected to such blocks in our network can still maintain normal synchronization. On the contrary, after attacks on Tron, RSK, and Klaytn, their block execution time exceeds their block generation interval. This leads to difficulties in normal synchronization for victim nodes subject to such blocks in the network.

• **RPCS weakness.** Contract Builderconstructs attack contracts for RPCS weaknesses that contain circular hash calculations to consume CPU resources of nodes. Fig. 11 shows the attack contract constructed by DoSDET for Ethereum.

Weakness detection. According to the output of DoSVER, the *payload* parameter is set to 10,000 for the attack contracts of Ethereum, Tron, Klaytn, and RSK, and the *payload* parameter is set to 1,400 for the Solana attack contract. Fig. 12 shows the impact of attack using RPCS on the system block synchronization rate of the five blockchains. It shows that the block synchronization slowdown on RSK, Tron, Ethereum, Klaytn, and Solana reached the highest of 24.0%, 38.7%, 32.0%, 36.8%, and 1.2%, respectively. Our results indicate that RPCS weakness has a relatively significant impact on the system synchronization rate of Ethereum, Tron, Klaytn, and

```
contract InlineDiv {
  function attack() returns(uint res, uint256 payload) {
    assembly {
      let x:= 100
      let y:= 2
      for {let i:= 0} lt(i, payload) {i := add(i, 1)} {
        res := div(x,div(x,div(...div(x,div(x,y))...)))}}}}
```

Figure 10: Inline assembly contracts utilizing DIV instructions

Table 9: Results of MP attack

Blockchain	Ethereum		Tron		RSK		Klaytn	
Attack contracts	SMOD	EXP	EXTCODESIZE	SLOAD	RETURNDATASIZE	DIV	SHR	DIV
Block execution time (s)	10.36	10.19	9.12	8.44	10.74	9.05	4.67	4.13

RSK, while it has almost no impact on Solana. After analysis, we believe that this is due to Solana's more comprehensive restriction policy on RPC services and the advantage that its unique historical proof consensus mechanism gives to the verifier.

Answer to RQ2: DoSDET *can effectively synthesize attacks against weaknesses and verify the existence of weaknesses, and it has discovered multiple DoS weaknesses in nine blockchains (cf. Table 1).*

6.4 RQ3: Accuracy of DoSVER

To analyze whether DoSVER has FN and FP, we construct all the attack scenarios analyzed by DoSVER to launch real attacks and check the correctness of DoSVER's results by observing whether the attacks are effective. Since DoSDET automatically synthesizes real attacks for the scenarios where DoSVER determines the existence of potential weaknesses, we only need to manually attack for the scenarios where DoSVER determines that no potential weaknesses exist. After the attack, the effectiveness of the attack is evaluated based on the metrics in §5.5 and compared with DoSVER's results. Eventually, we find no FP and only FN in the results of DoSVER and label all false negatives by ■ in Table 3.

Specifically, for the nine blockchains, DoSVER one detects 162 attack scenarios, of which 91 scenarios have potential weaknesses and 71 scenarios do not have potential weaknesses. In the actual attacks conducted against scenarios where DoSVER determines the existence of potential weaknesses, all the attacks are able to effectively affect the blockchain performance, i.e., no FP. In the actual attacks conducted against scenarios where DoSVER determines that no potential weaknesses exist, there are 9 scenarios where the attacks can effectively affect the blockchain performance, i.e., 9 FNs. We find that all of these FNs appeared in scenarios where the number of attackers is low, and the strength of attacks is low. In such attack scenarios, the attackers need to keep on initiating more transactions for a long time before they can exhaust the victim's resources, which can cause DoSVER's search space to become very large. As a result, DoSVER struggles to find the target trail in an acceptable amount of time, leading to FN. However, these FN ultimately did not affect the discovery of DoS weaknesses on the nine blockchains since DoSVER can automatically detect stronger attack scenarios during timeouts and DoSDET further validates DoSVER's results.

Answer to RQ3: *Out of the 162 attack scenarios verified by* DoSVER*, we find no FP and 9 FNs.*

6.5 RQ4: Overhead

Table 10 shows the time overhead of the proposed approach. Since only for blockchains with $\mathcal{P}_{MP}$, Step ② needs to perform additional probe insertion instead of just compiling the source code to generate the client, and Step ③ needs to synchronize a sufficient number of historical transactions to collect the t/c, m/c, and io/c ratios of the instructions, we consider the time overhead in this case separately. On average, it takes 274.6s for DoSVER to discover potential DoS weaknesses, and 519.1s for DoSDET to synthesize and validate attacks against weaknesses other than MP (8.8×10^4s for blockchains with MP weaknesses). Compared to manually analyzing the mechanism and source code of the blockchain and performing weakness detection, our method saves much time. For blockchains without $\mathcal{P}_{MP}$, the most time-consuming steps of DoSDET are compiling the source code (Step ②) and launching the attack (Step ④), which account for 95.03% of the overall execution time. For blockchains with $\mathcal{P}_{MP}$, the process of synchronizing historical transactions (Step ③) consumes 99.57% of the overall execution time, however, synchronizing historical transactions is an un-omitted process in order to collect pricing metrics for all instructions in the real world, so this overhead is unavoidable.

Answer to RQ4: *Our Our approach discovers weaknesses other than MP in an average of 766.7s, and spends an average of 24.4 hours against blockchains with $\mathcal{P}_{MP}$.*

```
1  contract DetRPC {
2    function exhaustCPU(uint256 payload) returns(bool) {
3      bytes32 var = 0xf...f;
4      for (uint256 i=0; i<payload; ++i){
5        var = keccak256(abi.encodePacked(var));}
6      return true;}}}
```

Figure 11: RPCS attack contract

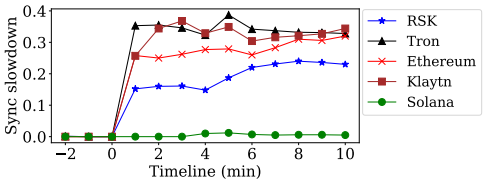


Figure 12: Block sync slowdown under RPCS attack

7 Weakness attribution

We further expose the design issues behind DoS weaknesses through CWE [8]. While some prior work has been done to identify smart contract vulnerabilities through CWE classification [41, 58], we utilize this approach for the first time to provide insights against system-level DoS weaknesses. CWE presents a weakness hierarchy consisting of *Pillar*, *Class*, *Base*, and *Variant* in a manner that ranges from abstract to specific, with *Pillar* nodes serving as the root of a rooted tree. We leverage blockchain characteristics to expand on CWE's resource-related weaknesses' descriptions to dissect DoS weaknesses in association with the resource model.

CWE-664: *Improper Control of a Resource Through its Lifetime.* The resource management mechanism maintained by the blockchain system throughout its life cycle is flawed.

Table 10: Time overhead of our approach.

	Min	Max	Mean
Overrall time cost of DoSVER	7.3s	587.4s	247.6s
Step ①: Info Collector	16.2s	31.4s	22.3s
Step ②: Probe Inserter	62.6s (87.3s)	228.2s (284.2s)	153.7s (189.4s)
Step ③: Contract Builder	2.2s (17.2h)	7.1s (31.4h)	3.5s (24.4h)
Step ④: Attack Detector	78.9s	621.4s	339.6s
Overrall time cost of DoSDET	159.9s (17.2h)	888.1 (31.5h)	519.1 (24.4h)

Since Step ② and Step ③ require additional steps for blockchains with $\mathcal{P}_{MP}$, we show the overhead of detection against such blockchains alone in parentheses.

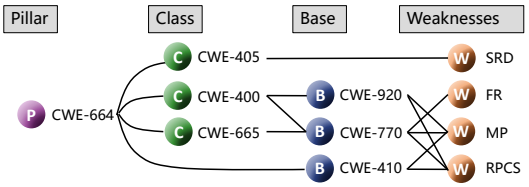


Figure 13: DoS weaknesses attribution dendrogram

CWE-400: *Uncontrolled Resource Consumption.* The defects in blockchain's resource management allow attackers to affect the allocation or consumption of limited resources.
CWE-405: *Asymmetric Resource Consumption (Amplification).* Blockchain systems have exploitable mechanisms that allow users to consume resources from other users.
CWE-665: *Improper Initialization.* The initialization resources provided by the blockchain system for new accounts may lead to security implications.
CWE-410: *Insufficient Resource Pool.* The computing resources and storage resources of blockchain nodes are insufficient to handle peak transactions.
CWE-770: *Allocation of Resource Without Limits or Throttling.* Improper resource allocation, resource restrictions, or other protection mechanisms that exist in the blockchain allow malicious users to violate expected security policies.
CWE-920: *Improper Restriction of Power Consumption.* Blockchain systems operate in a resource-limited environment without properly limiting its consumption of resources.

Based on above extensions, we analyze and attribute DoS weaknesses. We use CWE hierarchy to trace the four DoS weaknesses to the CWE-664 pillar node and report the resulting rooted tree in Fig. 13. We detail the software weaknesses that lead to various vulnerabilities in Appendix K. We have reported the newly discovered weaknesses to the corresponding development teams and proposed mitigations for each weakness by analyzing the design issues that led to each weakness (cf. Appendix L).

8 Discussion and Limitation

Our work facilitates the automation of discovering DoS weaknesses in the blockchain execution layer. For researchers engaged with multiple blockchains, our tools reduce the need for extensive manual learning for each platform by automating critical tasks such as model verification, contract generation, and actual attack simulation. In addition, DoSVER and DoSDET can work independently in our approach. For designers focused on a single blockchain, familiar with its usage and contract development, DoSDET may be unnecessary. However, they can leverage DoSVER for rapid initial assessments during the design and modification of blockchain resource models. DoSVER alone offers high accuracy (see Section 6.4), enabling designers to formally analyze potential DoS weaknesses in resource model design within seconds or minutes.

Since this work focuses on DoS weaknesses in the blockchain resource model, it cannot discover network layer DoS [70], memory pool blocking DoS [45], and incentive-based DoS [49] weaknesses. Moreover, due to blockchain's high concurrency and real-time nature, DoSVER inevitably faces the impact of state space explosion. Although we utilize uncritical configurations (§4) and symbolic search (Appendix H) for mitigation, it's still possible to generate false negatives and false positives due to timeout. Since verification scenarios with the lower number of attackers and attack strength require longer checking time, the results may be inaccurate due to timeouts in such scenarios. Nevertheless, DoSVER can verify multiple attack scenarios and discover potential weaknesses through stronger attack scenarios. Moreover, since the formalism's time representation cannot be precisely aligned with physical time, DoSVER identifies weaknesses but cannot measure the impact extent of DoS attacks on the system. To address this, we introduce the attack synthesis framework DoSDET, designed to work in tandem with DoSVER to execute actual attacks for validation and impact measurement. However, DoSDET's primary goal is to utilize the attacks to validate the existence of weaknesses. Thus, we only design to allow DoSDET to generate practical attacks without finding the optimal attack since designing the optimal attack involves complexities such as real-time analysis and strategy selection. Furthermore, the goal of this paper is to automatically discover whether the four known types of weaknesses exist on various blockchain systems rather than discover unknown new types of weaknesses. Therefore, after new DoS attack methods are discovered in the future, the tool needs to be manually updated to support new weakness discovery. The above limitations threaten effectiveness, and we treat them as future work to contribute more to the field.

9 Related work

As a work dedicated to DoS vulnerabilities, we have focused mainly on other similar studies, and the following are the DoS-related works we reviewed.
DoS attack analysis. Lee et al. analyze resource drain attacks on EOS, proposes metrics to evaluate the attack effects and discusses possible mitigations[42]. Chen et al. summarize various DoS attacks on Ethereum, analyze the vulnerabilities that caused the attacks, and propose defense methods[24]. Perez et al. discover inconsistencies in the pricing of the instructions in Ethereum[54], and lead to the metrics we used to evaluate the effectiveness of MP attacks. Chen et al. do an in-depth study on MP attacks on Ethereum, and propose an adaptive gas cost mechanism for ethereum to defend against this DoS attacks[25]. In addition to these, many researchers have also conducted detailed analysis of DoS attacks on blockchains[34, 46, 55]. However, these approaches all rely on expert manual analysis, and it is difficult to automate the analysis process.
Vulnerability discovery. Grech et al. present MadMax: a static program analysis technique to automatically detect gas-focused vulnerabilities[33]. Jiang et al. propose ContractFuzzer to find security vulnerabilities in Ethereum smart contracts[38]. Feng et al. present SmartScopy, which not only discovers vulnerabilities, but also synthesizes adversarial smart contracts to exploit vulnerable smart contracts[31]. In addition to the above, there are many excellent studies on blockchain vulnerability discovery. SmartCheck uses static analysis technology to find 21 kinds of problems from smart contracts[63]. Oyente[48], Manticore[50], Osiris[64], MAIAN[51] and MythX[3] both leverages symbolic execution to discover vulnerabilities with different implementations and focus on different kinds of vulnerabilities. However, these works are all for smart contract vulnerability discovery, and most of them focus only on

Ethereum. In contrast, our research is system-level vulnerability discovery for multiple blockchains.

10 Conclusion

DoS attacks on blockchains can cause huge economic losses and undermine users' confidence in blockchains. We provide the first systematic study of diverse DoS weaknesses in multiple blockchains. Specifically, we present a tool called DoSVER based on formal verification aimed at reasoning about DoS weaknesses. Then, we design an automated attack synthesis framework called DoSDET that can be applied to multiple blockchains to verify and discover DoS weaknesses automatically. We apply our approach to 9 different and representative blockchains and uncover DoS weaknesses in them. In addition, we dissect the design issues behind blockchain DoS weaknesses from the perspective of the resource model.

Acknowledgments

The authors thank the anonymous reviewers for their constructive comments. This work is partly supported by National Natural Science Foundation of China (No. 62332004), Sichuan Provincial Natural Science Foundation for Distinguished Young Scholars (No. 2023NSFSC1963), Hong Kong RGC Projects (No. PolyU15222320, PolyU15231223), and Hong Kong ITF Project (No. PRP/005/23FX).

References

[1] 2016. Ethereum faces another dos attack. https://cryptohustle.com/ethereum-faces-another-dos-attack/.
[2] 2016. Transaction spam attack: Next Steps. https://blog.ethereum.org/2016/09/22/transaction-spam-attack-next-steps/.
[3] 2019. Smart contract security tool for Ethereum. https://mythx.io/.
[4] 2021. Binance Markets. https://www.binance.com/en/markets.
[5] 2021. Resource Model. https://developers.tron.network/docs/resource-model.
[6] 2021. Virtual Machine Introduction. https://developers.tron.network/docs/vm-introduction.
[7] 2022. *Coindataflow.* https://coindataflow.com/en
[8] 2022. Common Weakness Enumeration. https://cwe.mitre.org/.
[9] 2022. Cryptocurrencies - Total Market Capitalization. https://sc.macromicro.me/collections/3785/crypto/33112/cryptocurrency-total-market-cap.
[10] 2022. Deploy the FullNode or SuperNode. https://developers.tron.network/docs/fullnode.
[11] 2022. *GPT-4: The Next Breakthrough in Language Modeling.* https://openai.com/gpt-4/
[12] 2022. Hardware requirements. https://developers.rsk.co/rsk/node/install/requirements/.
[13] 2022. Klaytn vs. Ethereum - Differences. https://forum.klaytn.com/t/klaytn-vs-ethereum-differences/20.
[14] 2022. The Maude System. http://maude.cs.illinois.edu/w/index.php/The_Maude_System.
[15] 2022. *Metallicus.* https://developer.wax.io/dapps/supported-operating-systems/
[16] 2022. Overview and architecture. https://docs.telos.net/evm/about-ethereum-virtual-machine/overview-and-architecture.
[17] 2022. Prequisites. https://docs.telos.net/eosio-docs/getting-started/developer-environment/1.1-prequisites.
[18] 2022. Supported Operating Systems. https://developer.wax.io/dapps/supported-operating-systems/.
[19] 2022. System Requirements. https://docs.klaytn.com/node/endpoint-node/system-requirements.
[20] 2022. Turing complete. https://developers.rsk.co/rsk/architecture/turing-complete/.
[21] 2023. Today's Cryptocurrency Prices by Market Cap. https://coinmarketcap.com/.
[22] Abraão Aires Urquiza, Musab A. AlTurki, Max Kanovich, Tajana Ban Kirigin, Vivek Nigam, Andre Scedrov, and Carolyn Talcott. 2019. Resource-Bounded Intruders in Denial of Service Attacks. In *IEEE CSF (2019).*
[23] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. 2013. *Systems and software verification: model-checking techniques and tools.* Springer Science & Business Media.
[24] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. 2019. A survey on Ethereum systems security: Vulnerabilities, attacks and defenses. *arXiv* (2019).
[25] Ting Chen, Xiaoqi Li, Ying Wang, Jiachi Chen, Zihao Li, Xiapu Luo, Man Ho Au, and Xiaosong Zhang. 2017. An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks. In *ISPEC (2017).*
[26] Ting Chen, Zihao Li, Yufei Zhang, Xiapu Luo, Ang Chen, Kun Yang, Bin Hu, Tong Zhu, Shifang Deng, Teng Hu, et al. 2019. Dataether: Data exploration framework for ethereum. In *IEEE ICDCS (2019).*
[27] Ting Chen, Zihao Li, Yuxiao Zhu, Jiachi Chen, Xiapu Luo, John Chi-Shing Lui, Xiaodong Lin, and Xiaosong Zhang. 2020. Understanding ethereum via graph analysis. *ACM TOIT* (2020).
[28] Alexandre David, Jacob Illum, Kim G Larsen, and Arne Skou. 2018. Model-based framework for schedulability analysis using UPPAAL 4.1. In *Model-based design for embedded systems.*
[29] Conrado Daws and Sergio Yovine. 1995. Two examples of verification of multirate timed automata with Kronos. In *IEEE RTSS.*
[30] Ardit Dika. 2017. *Ethereum smart contracts: Security vulnerabilities and security tools.* Master's thesis. NTNU.
[31] Yu Feng, Emina Torlak, and Rastislav Bodik. 2019. Precise attack synthesis for smart contracts. *arXiv* (2019).
[32] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. 2022. ETainter: Detecting Gas-Related Vulnerabilities in Smart Contracts. In *ISSTA.*
[33] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *OOPSLA* (2018).
[34] Richard Greene and Michael N Johnstone. 2018. An investigation into a denial of service attack on an ethereum network. (2018).
[35] Ningyu He, Ruiyi Zhang, Haoyu Wang, Lei Wu, Xiapu Luo, Yao Guo, Ting Yu, and Xuxian Jiang. 2021. EOSAFE: Security Analysis of EOSIO Smart Contracts. In *USENIX Security.*
[36] Z. He, Z. Li, A. Qiao, X. Luo, X. Zhang, T. Chen, S. Song, D. Liu, and W. Niu. 2024. Nurgle: Exacerbating Resource Consumption in Blockchain State Storage via MPT Manipulation. In *IEEE SP.*
[37] Zheyuan He, Zhou Liao, Feng Luo, Dijun Liu, Ting Chen, and Zihao Li. 2022. TokenCat: detect flaw of authentication on ERC20 tokens. In *IEEE ICC.*
[38] Bo Jiang, Ye Liu, and WK Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *ASE (2018).*
[39] Max Kanovich, Tajana Ban Kirigin, Vivek Nigam, Andre Scedrov, and Carolyn Talcott. 2016. Timed multiset rewriting and the verification of time-sensitive distributed systems. In *Formal Modeling and Analysis of Timed Systems.*
[40] Max Kanovich, Tajana Ban Kirigin, Vivek Nigam, Andre Scedrov, and Carolyn Talcott. 2017. Time, computational complexity, and probability in the analysis of distance-bounding protocols. *Journal of Computer Security* (2017).
[41] Jong-Hoon Lee, SeongHo Yoon, and Hyuk Lee. 2022. Swc-based smart contract development guide research. In *IEEE ICACT (2022).*
[42] Sangsup Lee, Daejun Kim, Dongkwan Kim, Sooel Son, and Yongdae Kim. 2019. Who Spent My EOS? On the (In)Security of Resource Management of EOS.IO. In *USENIX WOOT (2019).*
[43] John Levine. 2009. *Flex & Bison: Text Processing Tools.* " O'Reilly Media, Inc.".
[44] Kai Li, Jiaqi Chen, Xianghong Liu, Yuzhe Tang, XiaoFeng Wang, and Xiapu Luo. 2021. As strong as its weakest link: How to break blockchain dapps at RPC service. In *NDSS (2022).*
[45] Kai Li, Yibo Wang, and Yuzhe Tang. 2021. DETER: Denial of Ethereum Txpool SERvices. In *ACM CCS (2021).*
[46] Xiaoqi Li, Peng Jiang, Ting Chen, Xiapu Luo, and Qiaoyan Wen. 2020. A survey on the security of blockchain systems. *FGCS* (2020).
[47] Feng Luo, Ruijie Luo, Ting Chen, Ao Qiao, Zheyuan He, Shuwei Song, Yu Jiang, and Sixing Li. 2024. Scvhunter: Smart contract vulnerability detection based on heterogeneous graph attention network. In *ICSE.*
[48] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *ACM CCS.*
[49] Michael Mirkin, Yan Ji, Jonathan Pang, Ariah Klages-Mundt, Ittay Eyal, and Ari Juels. 2020. BDoS: Blockchain Denial-of-Service. In *ACM CCS.*
[50] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *ASE.*
[51] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *ACSAC.*
[52] Carlos Olarte and Peter Csaba Ölveczky. 2024. Timed Strategies for Real-Time Rewrite Theories. In *WRLA.*
[53] Peter Csaba Ölveczky and José Meseguer. 2002. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science* (2002).
[54] Daniel Perez and Benjamin Livshits. 2019. Broken metre: Attacking resource metering in EVM. *arXiv* (2019).
[55] Mohsin Ur Rahman. 2020. Scalable role-based access control using the eos blockchain. *arXiv* (2020).
[56] Sergio Ramírez, Miguel Romero, Camilo Rocha, and Frank Valencia. 2018. Real-Time Rewriting Logic Semantics for Spatial Concurrent Constraint Programming. In *WRLA.*

[57] Camilo Rocha, José Meseguer, and César Muñoz. 2017. Rewriting modulo SMT and open system analysis. *J LOG ALGEBR METHODS* (2017).
[58] Mirko Staderini, Caterina Palli, and Andrea Bondavalli. 2020. Classification of ethereum vulnerabilities and their propagations. In *IEEE BCCA*.
[59] Jinlei Sun, Song Huang, Changyou Zheng, Tingyong Wang, Cheng Zong, and Zhanwei Hui. 2022. Mutation testing for integer overflow in ethereum smart contracts. *Tsinghua Science and Technology* (2022).
[60] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. 2023. When GPT Meets Program Analysis: Towards Intelligent Detection of Smart Contract Logic Vulnerabilities in GPTScan. *arXiv* (2023).
[61] Zhiyuan Sun, Xiapu Luo, and Yinqian Zhang. 2023. Panda: Security analysis of algorand smart contracts. In *USENIX Security*.
[62] Andrei Tara, Kirill Ivkushkin, Alexandru Butean, and Hjalmar Turesson. 2019. The evolution of blockchain virtual machine architecture towards an enterprise usage perspective. In *Computer Science On-line Conference*.
[63] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. Smartcheck: Static analysis of ethereum smart contracts. In *WETSEB*.
[64] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for integer bugs in ethereum smart contracts. In *ACSAC (2018)*.
[65] Liangmin Wang, Victor S. Sheng, Boris Düdder, Haiqin Wu, and Huijuan Zhu. 2023. Security and privacy issues in blockchain and its applications. *IET Blockchain* (2023).
[66] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* (2014).
[67] Youngseok Yang, Taesoo Kim, and Byung-Gon Chun. 2021. Finding consensus bugs in ethereum via multi-transaction differential fuzzing. In *USENIX OSDI*.
[68] Kunsong Zhao, Zihao Li, Jianfeng Li, He Ye, Xiapu Luo, and Ting Chen. 2023. Deepinfer: Deep type inference from smart contract bytecode. In *ESEC/FSE*.
[69] Shuyu Zheng, Haoyu Wang, Lei Wu, Gang Huang, and Xuanzhe Liu. 2020. VM Matters: A Comparison of WASM VMs and EVMs in the Performance of Blockchain Smart Contracts. *arXiv* (2020).
[70] Liyi Zhou, Kaihua Qin, and Arthur Gervais. 2021. A2mm: Mitigating frontrunning, transaction reordering and consensus instability in decentralized exchanges. *arXiv* (2021).

A Differences between blockchain VMs

Depending on the developer's design, these VMs will execute smart contract bytecode based on either a custom instruction set or a standard instruction set. We show the nine blockchain VMs and instruction set differences in Table 11.

The Ethereum Virtual Machine (EVM) adopts a stack-and-memory model with a 256-bit instruction word size. This model provides access to the program stack, a register space, an expandable temporary memory and a permanent storage [36, 62]. The EVM supports an instruction set of 150 8-bit opcodes to allow the creation of Turing-complete programs that support smart contract execution [67]. The TRON Virtual Machine (TVM) is a lightweight VM compatible with the EVM environment and has a unique virtual memory mechanism that reduces memory usage [6]. The RSK Virtual Machine (RVM) and Klaytn Virtual Machine (KLVM) are based on EVM with a unique design and are therefore compatible with but different from EVM. The gas of RVM is paid in RBTC, which is one-to-one pegged to Bitcoin. KLVM uses platform-defined values for gas prices instead of gas prices from users [13].

Table 11: VMs used by nine blockchains

Blockchain	Virtual machine	Instruction set	Data types	
			Integers	Floating Point
Ethereum	EVM	Custom instructions (ADD, SSTORE, SHA3, CALL etc.)	8 up to 256 bits	Not fully supported
Tron	TVM			
RSK	RVM			
Klaytn	KLVM			
EOS	EOS-VM	Standard instructions	32/64 bit	32/64 bit
WAX	EOS-VM			
BOSCore	EOS-VM			
Telos	tEVM			
Solana	Neon			

WAX and BOSCore are side chains of EOS, so they directly use EOS-VM, the VM of EOS. EOS-VM relies on the Web Assembly Virtual Machine (WAVM), a stack-based model, and then implements a WebAssembly (WASM)-based contract engine. Telos is also a sidechain of EOS, but its VM called tEVM is implemented as a smart contract on the blockchain [16]. Table 11 shows the technical differences between the VMs of the nine blockchains in terms of the instruction set and data types [6, 13, 20, 69].

B Transaction Specification Supplement

B.1 Other rules

The purpose of a resource-related DoS attack is for the attacker to use a small amount of their own resources to cause significant depletion of the victim's quantifiable resources or the machine resources of the victim node. The essence of this type of DoS attack is to maximize resource consumption through transactions, achieved by generating and deploying contracts with high resource usage and invoking them as frequently as possible. Consequently, when quantifying the impact of this attack transaction on the system using the verifier, only the consumption of resources by the transaction needs consideration, without requiring a specific focus on the operational semantics of the target contract.

B.2 Example of transaction execution rule

We use a concretization of the **CRE** rule in Fig. 3 as an example to aid understanding. Note that for easy observation, the numbers in this example are assumed to be simple integers. Suppose a user named Bob wants to deploy a contract called Hi in the Ethereum system at time T. Bob currently has 100 gas in his account, and the transaction tx required to deploy the contract Hi consumes a transaction fee of 10 gas. Therefore, after the transaction is completed, Bob will have 90 gas remaining. Additionally, the node currently has 50% of CPU resources and 500kb of memory resources available. However, the processing of transaction tx requires 1% of CPU and 2kb of memory resources. As a result, the transaction will continue to occupy the corresponding resources until it is completed. The system takes 1 second to execute the transaction, so after the transaction is executed, the Ethereum system time becomes $T+1$, and a new contract Hi is added to the blockchain. At this point, a **CRE** rule describing the execution process of transaction tx can be instantiated as shown in Fig. 15.

Transaction Terminated Rule:

$$\begin{bmatrix} Time@T \\ \mathsf{S}_i(tx, R_{tx}, c_{id})@T \\ \mathsf{User}(tx, r_c - r)@T_1 \\ \mathsf{Owner}(tx, r_o - r)@T_2 \\ \mathsf{Node}(R)@T_3 \end{bmatrix} \rightarrow \begin{bmatrix} Time@T \\ \mathsf{User}(tx, r_c - r)@T \\ \mathsf{Owner}(tx, r_o - r)@T \\ \mathsf{Node}(R)@T \end{bmatrix}$$

Transaction Availability Rules:

$$\begin{bmatrix} Time@T \\ \mathsf{Node}(R_{tx} - R)@T \\ \mathsf{Av}(tx)@T_1 \end{bmatrix} \rightarrow \begin{bmatrix} Time@T \\ \mathsf{Node}(R_{tx} - R)@T \\ \mathsf{Bl}(tx)@T \end{bmatrix}$$

$$\begin{bmatrix} Time@T \\ \mathsf{Node}(R_{tx} + R)@T \\ \mathsf{Bl}(tx)@T_1 \end{bmatrix} \rightarrow \begin{bmatrix} Time@T \\ \mathsf{Node}(R_{tx} + R)@T \\ \mathsf{Av}(tx)@T \end{bmatrix}$$

Figure 14: Other Two Types of Rules

$$\begin{bmatrix} Time@T \\ \mathsf{S}_0(tx,[1,2],\oslash)@T_1 \\ \mathsf{User}(tx,100)@T_2 \\ \mathsf{Node}([50,500])@T_3 \\ \mathsf{N}(CRE)@T_4 \end{bmatrix} \xrightarrow[T_2,T_3,T_4 \le T]{T_1 \ge T} \exists Hi. \begin{bmatrix} Time@T \\ \mathsf{S}_1(tx,[1,2],Hi)@(T+1) \\ \mathsf{User}(tx,90)@T \\ \mathsf{Node}([49,498])@T \end{bmatrix}$$

Figure 15: CRE rule instantiation

B.3 The role of transaction $\mathcal{US}$

Without the Terminated $\mathcal{US}$, false DoS attacks could be found as traces where the transaction does not release the held NR even if the transaction is terminated due to insufficient QR. Similarly, without Rejected $\mathcal{US}$, Available $\mathcal{US}$ and Creatable $\mathcal{US}$, there would be traces where the facts $\mathsf{Av}(tx)@t$, $\mathsf{Bl}(tx)@t$ and $\mathsf{Rej}(tx)@t$ are not updated according to the level of resources of the transaction. For example, a trace would exist with a configuration where $\mathsf{Rej}(tx)$ is present, although the transaction tx has enough QR.

C Proof

Given a verification scenario $\mathcal{V}$, with all the notation as per Definition 4.6, we construct an instance of the critical trace reachability problem, $\mathcal{D}$, such that $\mathcal{D}$ has a solution trace $\mathcal{T}$ iff $\mathcal{V}$ is vulnerable to a DoS attack as per witness trace $\mathcal{T}$. This follows easily from the definition of the problem $\mathcal{D}$, by inspecting the transaction and users rules, uncritical and goal configurations.

We set the critical trace reachability problem $\mathcal{D}$ as follows. We set MSR rules $\mathcal{R}$ to consist of users rules $\mathcal{U}_j$, $1 \le j \le m$, and rules of transaction theories $\mathcal{T}_i$, $1 \le i \le n$. The initial coniguration of $\mathcal{D}$ is set to be S_I, the inital configuration of verification scenario $\mathcal{V}$. Also, the goal of $\mathcal{D}$ is specified by the goal of $\mathcal{V}$:

$$\langle\, Time@T, \mathsf{Bl}(s_i)\lambda)@T', \quad T - T' \ge d_{min} \rangle.$$

We set uncritical configuration specification $\mathcal{US}$ of $\mathcal{V}$ as $\mathcal{US}_{\mathcal{V}}$, i.e., to consist of protocol $\mathcal{US}_{es}$ of $\mathcal{A}_1, \ldots, \mathcal{A}_n$. Note that as per transaction uncritical configuration specifications, $\mathcal{US}$ of $\mathcal{D}$ contains:

$$\langle\, \mathsf{R}(S_{id},\ r_m^{S_{id}})@T, \mathsf{Av}(S_{id})@T' \mid T > T' \rangle \text{ and}$$

$$\langle\, \mathsf{R}(S_{id},\ R+1+r_m^{S_{id}})@T, \mathsf{Bl}(S_{id})@T' \mid T > T' \rangle$$

for the auxiliary flags, Bl, Rej, used for modelling insufficient resources.

D Information to be Collected in Step ①

Fig. 16 shows the prompts for ❶ and ❷. Table 12 shows the information that can be collected by the Info Collector.

```
System: System: same as in Fig. 6.
Question ❶
What are the supported smart contract languages for
      [%Blockchain Type%]? Answer only the name.
Question ❷
Does[%Blockchain Type%] have a [%Prerequisite Type%] mechanism? Answer
       only "Yes" or "No".
```

Figure 16: Prompt for question ❶ and ❷

E Probe insertion algorithm

When finding the target subtree, the parser will output an instrumentation information formalized by a quadruple (P, O, S, E), where:

P is the path of the file being parsed; O is the function of the current code segment; S/E is the position to be inserted at the beginning/end of the code fragment. All instrumentation information is collected as a set *location*.

After obtaining the *location*, Probe Inserter use the process shown in Algorithm 1 to insert probes that obtain runtime information. The `judge_Language()` method analyzes the source code language. The `add_startTime()` method sets the file pointer at the specific position of the specific file according to *path* and *start*, and uses the corresponding programming language according to the *language* to insert a variable T_{start} to mark the start time at the position pointed. The `add_opcode_endOperate()` method will insert a variable T_{end} marking the end time after setting the file pointer in the same way, and then increase the statement of output *operate* and the cost of this opcode and time change $\Delta T = T_{end} - T_{start}$ according to the *language*. The `add_block_endOperate()` method also inserts T_{end} and adds a statement that outputs *operate* and ΔT, and additionally adds a statement that outputs the QR spent executing the entire block.

Algorithm 1 probe insertion algorithm

Input: Blockchain source code: $Source_{ori}$
Instrumentation information set: *location*
Output: Instrumented blockchain source code

```
language ← judge_Language(Source_ori)
for each (P, O, S, E) in location do
    if O = process block then
        Source_ori.add_startTime(language, P, S)
        Source_ori.add_block_endOperate(language, P, E, O)
    else                                          ▷ opration is to execute opcode
        Source_ori.add_startTime(language, P, S)
        Source_ori.add_opcode_endOperate(language, P, E, O)
    end if
end for
```

F Simulation attack steps

Note that we turn off the compiler's optimized compilation function when compiling the contract and we transfer enough QRs to attacker accounts created during detection of SRD and MP to keep them attacking.

SCP Resource drain attack. Suppose that C_1/C_2 is the CPUD/RAMD contract: **Step 1)** Create a victim account V. **Step 2)** V deploys C_1/C_2 and grants C_1/C_2 `eosio.code` permission. **Step 3)** Create

Table 12: Details of Commands and Information

Method	Information	Details
Direct QA	Compile Code Comd	compile blockchain and tools source code.
	Synchronization Cmd	synchronize blockchain nodes with mainnet.
	Building Cmd	build a blockchain private chain.
	Creation Cmd	create accounts in the blockchain.
	Transfer Cmd	transfer money in the blockchain.
	Compile Contract Cmd	compile smart contract.
	Deployment Cmd	deploy smart contracts in the blockchain.
	Invoking Cmd	invoke function or action.
	Accounts Cmd	get the account information.
	Transaction Cmd	get transactions information.
Problem decomposition	$\mathcal{P}_{FR}$	the amount of free sources in this blockchain.
	$\mathcal{P}_{RPCS}$	RPC interface with free contract execution capability.
	$\mathcal{P}_{SRD}$	Information to detect if the blockchain has `eosio.code` permission and defer action.
	$\mathcal{P}_{MP}$	Information to detect if the blockchain has opcode.
	Contract Language	contract language of this blockchain.

attacker accounts $A_1, \ldots, A_n$ according to the number of attackers output by DoSVER, A_i $(1 \leq i \leq n)$ continuously calls C_1, captures the CPU consumption of V and A and calculates the ratio. **Step 4)** Create attacker accounts $A_1, \ldots, A_n$ according to the number of attackers output by DoSVER, A_i $(1 \leq i \leq n)$ keeps calling C_2, capturing RAM consumption of V.

Free resource attack. Suppose that C_1 is the RAMD contract: **Step 1)** Create a victim account V. **Step 2)** V deploys C_1 and grants C_1 `eosio.code` permission. **Step 3)** Create attacker accounts $A_1, \ldots, A_n$ according to the number of attackers output by DoSVER, A_i $(1 \leq i \leq n)$ keeps calling C_1, capturing RAM consumption of V.

Mispricing attack. Suppose that C_1 is the ancillary contracts, C_2 and C_3 are two attack contracts that utilize different instructions. T_1, T_2, and T_3 are the transactions that invoke these three contracts, respectively: **Step 1)** Create attacker accounts $A_1, \ldots, A_n$ according to the number of attackers output by DoSVER. **Step 2)** A_i $(1 \leq i \leq n)$ continuously calls C_1, monitors the execution time of the current block, and finds the block filled with T_1 to calculate its execution time. **Step 3)** Use C_2 for the second step to calculate the execution time of the block full of T_2. **Step 4)** Use C_3 to perform the second step to calculate the execution time of the block full of T_3.

RPC attack. For this evaluation, we set up a fully synchronized mainnet full node in advance on other servers with the same configuration as the experimental server as the victim node, and opened the RPC of the victim node by default. DoSDET starts an attack node and a measurement node, both running Curl to send RPC requests. Suppose that V, A, and M are the victim node, attack node, and measurement node, respectively, and C is the attack contract: **Step 1)** Measure the current initial block height H_0 of the victim node by querying the block height of the PRC operation (e.g., `eth_getBlockNumber`), and continuously monitor its block height and record it as H_v, and additionally monitor the block height H_r of a regular mainnet node. **Step 2)** The attacking node initiates an operation to call contract C to the victim node through a speculative RPC operation for 10 minutes. **Step 3)** Calculate the block synchronization deceleration metric [44] by $H_r(i) - H_v(i)/H_r(i) - H_0$, where $H_v(i)$ and $H_r(i)$ represent the block height i minutes after the block height reaches H_0.

G Circumventing RPC service protection mechanisms

Note that there are protection mechanisms in the RPC service: load balancing strategy, QR limit and rate limit. Since there is only one victim node in our experiment, the load balancing problem can be ignored. In fact, according to research [44], there are still effective ways to evade load balancing strategies in the presence of multiple nodes, so the way we use single-node experiments will not affect the accuracy of our testing of RPC DoS attack effects. In addition, [44] also found that the Gas limit set by the current blockchain RPC service is at least 1.5 block QR, and when the *payload* in the `exhaustCPU` contract is 10000, its QR consumption is strictly lower than 1.5 block QR. Finally, since the rate limit in the RPC service is only for a single account, by creating 50 accounts to attack at the same time, we can make the attack frequency reach 100 times per second, evading the rate limit.

H Implementation

To deal with the problem that the trace is too long and leads to an enormous search space in the automatic verification, we use symbolic search to assist the implementation of our model. Note that although our model can verify four DoS attacks in principle, it is challenging to automate the verification in practice because the traces can be very long result in enormous search space. Fortunately, previous work [22] has demonstrated that symbolic search can be used to help with this type of verification. We use symbolic search through Rewriting Modulo SMT [57]. This allows us to perform symbolic search relying on the power of off-the-shelf SMT solvers. To achieve this, we consider the following two types of symbols:

- ***Time Symbols.*** Instead of using concrete values for global time, we use time symbols;
- ***Transaction Instance Symbols.*** We allow an attacker to create multiple transaction instances, not just one transaction, which represent bursts of attack behavior. However, we use symbolic values for the number of instances created.

Since attackers create transactions without a frequency limit, the search space will greatly increase, so we limit the attacker to only 2 transactions in a time unit window. By specifying values for blockchain-specific variables in §4.3 and §4.4, our model can concretize the corresponding resource models. Note that the execution time of a transaction in the blockchain should be considered jointly with its NR consumption R and QR consumption r. To this end, we construct an attack knowledge base based on the analysis of real attack contracts exploiting different vulnerabilities, which describes the strength of four DoS attacks with respect to R and r. We represent the attack strength in terms of the execution time t of real attack contracts, since the number of attack actions in the contracts is positively correlated with the execution time.

We represent the attack strength in terms of the execution time t of real attack contracts exploiting different vulnerabilities. For SRD, FR, and RPCS, we measure $t - r$ as well as $t - R$ curves for contracts with different attack intensities. For MP, we generalize the simulation-based Ethereum instruction execution time measurement framework in [25] to Tron, RSK, and Klaytn, and measure the execution time of each instruction with the corresponding r and R. Since the attack contract using MP repeats an instruction in large numbers, the execution time t of the attack contract can be approximated as the number of instruction repetitions multiplied by the execution time. After entering the type of vulnerability to be verified, the attack strength, and the number of attackers, our verification model simulates the corresponding attack use cases based on the attack knowledge base and starts verification.

I Other experimental results

I.1 SCP Resource drain weakness

CPU drain attack. Table 13 shows the effect of CPU drain attack when the number of attackers is 30. Table 14 shows the effect of CPU drain attack when the number of attackers is 50

I.2 Mispricing Weakness

In the four blockchains Etherum, Tron, RSK, and Klaytn, the m/c rate and io/c rate of the relevant instructions are shown in Fig. 17 and Fig. 18. It can be seen that in these blockchains, the m/c rate and

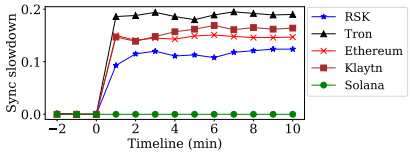


Figure 19: Results of the RPCS attack (attacker number is 50)

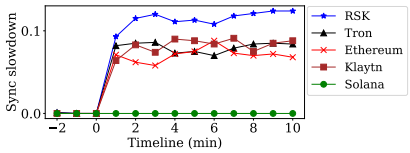


Figure 20: Results of the RPCS attack (attacker number is 30)

```
void ramconsume(name user,uint64_t payload){ //VA
  require_auth(user);
  eosio::transaction txn;
  const uint128_t sender_id = (((uint128_t)(
        current_time_point().sec_since_epoch()) << 64) | (
        uint128_t)(user.value));
  txn.actions.emplace_back(
  action(permission_level{get_self(),"active"_n},
  get_self(),
  "ramvuln"_n,
  std::make_tuple(user,payload)));
  txn.delay_sec = 0;
  txn.send(sender_id, get_self(),false);
}
[[eosio::action]]
void ramvuln(name user,uint64_t payload){
  require_auth(get_self());
  data_index data_table(get_first_receiver(),
        get_first_receiver().value);
  auto itr = data_table.find(0);
  if(itr != data_table.end()){
    uint64_t begin_idx = (itr->user) + 1,end_idx = (itr->user
          ) + payload;
    for(uint64_t i=begin_idx;i<=end_idx;i++){
      data_table.emplace(get_self(), [&]( auto& row ) {
      row.index = i;
      row.user = user.value;});}
    data_table.modify(itr,get_self(), [&]( auto& row ) {row.
          user = end_idx;});}
  return ;
}
```

Figure 21: Victim contract of RAMD attack

Table 13: Results of CPU drain attack when the attacker is 30

Blockchain	Victim's CPU			Attacker's CPU			CPU cost ratio		
	payload			payload			payload		
	50	100	1000	50	100	1000	50	100	1000
EOS	32817	63261	457199	18344	19080	19145	1.78	3.32	23.88
Telos	35191	69400	452966	19377	20036	20829	1.82	3.46	21.75
WAX	9664	15378	104717	16894	16825	15969	0.57	0.91	6.56
BOSCore	9004	13654	104937	16501	16929	15270	0.55	0.81	6.87

Table 14: Results of CPU drain attack when the attacker is 50

Blockchain	Victim's CPU			Attacker's CPU			CPU cost ratio		
	payload			payload			payload		
	50	100	1000	50	100	1000	50	100	1000
EOS	58428	112046	758399	32362	32308	31003	1.80	3.46	24.46
Telos	63377	114033	750414	32170	32687	32104	1.97	3.49	23.37
WAX	17419	24622	177204	26828	27458	26518	0.65	0.90	6.68
BOSCore	14362	22391	189155	25810	25866	25470	0.56	0.87	7.73

io/c rate of any one instruction do not appear to have several times difference compared to other instructions in the same blockchain, so for both memory and IO, these instructions do not appear to be improperly priced.

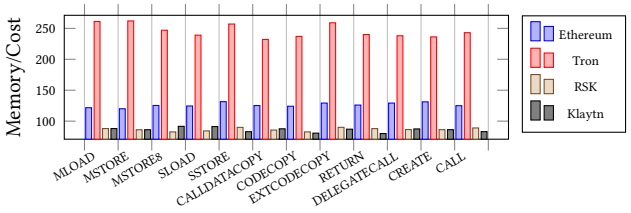


Figure 17: m/c rate of instructions for four blockchains

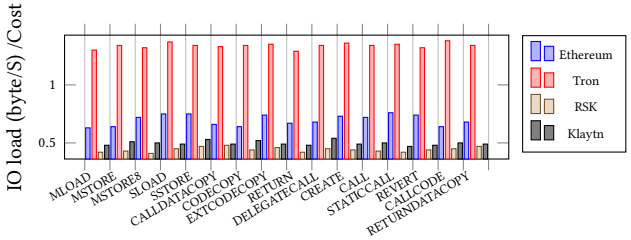


Figure 18: io/c rate of instructions for four blockchains

I.3 RPC services weakness

Fig. 19 and Fig. 19 show the impact of the RPC service attack on the blockchain system block synchronization rate for the number of attackers of 50 and 30, respectively. The results show a similar trend as when the number of attackers is 100. Solana receives almost no impact from the attack compared to the other three blockchains. As the number of attackers decreases, the impact of the RPC service attack on the other three blockchains is mitigated.

J RAMD contract

Fig. 21 shows the victim contract in a RAM drain attack, where the attacker can make the victim contract call the ramvuln() function to drain its own RAM by calling the ramcosume() function.

K Weaknesses Attribution

Resource Drain Weakness. Although the special deferred transaction mechanism of the blockchain represented by EOS improves the operability of transactions, it also introduces SRD to the blockchain system. For smart contracts with `eosio.code` permissions, attackers who use deferred transactions can easily consume their own low-cost QR to consume the contract owner's high-cost QR.

Free Resource Weakness. The mechanism introduced by some blockchains to allocate a fixed amount of free resources to users can encourage users to use, but it also brings convenience to malicious attackers and introduces a new type of vulnerability such as FR. Although the designers of the blockchain have limited the free amount to a certain extent, they obviously underestimated the power of the free mechanism combined with other attack methods. For example, an attacker can perform a CPU drain attack with

free resources, and can even drain the victim's QR with zero QR overhead.

Mispricing Weakness. Blockchains represented by Ethereum build the basis for smart contract operation by customizing instructions and limit resource abuse by setting costs for them, but the costs of these instructions are not proportional to the NR consumed. This inappropriate limitation results in MP, by exploiting MP instructions, attackers can consume their own lower-cost QR to consume NR of all blockchain nodes. Due to the limited resources of nodes, the throughput of the blockchain can drop significantly after suffering such an attack.

RPC Services Weakness. Speculative operations in blockchain RPC services can attract developers, but also allow attackers to trigger contract execution for free locally on the node where the RPC service is enabled. While the result of this attack leveraging RPCS does not affect the blockchain, the attacker can severely consume the limited NR of the target node through zero QR consumption, thereby slowing down its processing speed.

L Mitigation measures

For SRD weakness, a potential mitigation approach is to set a consumption cap for the contract over a period of time and blacklist callers that have excessive resource consumption. For FR weakness, a potential approach is to set an upper limit on the number of free resources that can be involved in a transaction fee to prevent an attacker from being able to completely cover the transaction fee with free resources. For MP weakness, a potential approach is to introduce an adaptive instruction pricing mechanism by considering the joint impact of major factors like SSD/memory/execution time. For RPCS weakness, a potential approach is to avoid tight coupling between the RPC service and other modules and adopt an access control mechanism to handle the incoming RPC requests.