SigVM: Enabling Event-Driven Execution for Truly Decentralized Smart Contracts

ZIHAN ZHAO, University of Toronto, Canada
SIDI MOHAMED BEILLAHI, University of Toronto, Canada
RYAN SONG, University of Toronto, Canada
YUXI CAI, University of Toronto, Canada
ANDREAS VENERIS, University of Toronto, Canada
FAN LONG, University of Toronto, Canada

This paper presents SigVM, a novel blockchain virtual machine that supports an event-driven execution model, enabling developers to build truly decentralized smart contracts. Contracts in SigVM can emit signal events, on which other contracts can listen. Once an event is triggered, corresponding handler functions are automatically executed as signal transactions. We build an end-to-end blockchain platform SigChain and a contract language compiler SigSolid to realize the potential of SigVM. Experimental results show that our benchmark applications can be reimplemented with SigVM in a truly decentralized way, eliminating the dependency on centralized and unreliable mechanisms like off-chain relay servers. The development effort of reimplementing these contracts with SigVM is small, i.e., we modified on average 3.17% of the contract code. The runtime and the gas overhead of SigVM on these contracts is negligible.

1 INTRODUCTION

Blockchain has become a revolutionary technology that powers decentralized ledgers. Ethereum, the second largest blockchain, introduces smart contracts, which further fuel blockchain innovations on real-world applications in various domains, including financial systems, supply chains, and health cares. A smart contract is a program operating on the blockchain ledger to encode customized transaction rules. Once deployed, the contract and the encoded rules are then faithfully executed and enforced by all participants of the blockchain platform, eliminating any potential counter-party risks in the future.

A Smart contract in Ethereum can generate *events*. Event is a record of data to log state changes or contract activities. To implement events, Ethereum Virtual Machine (EVM) [Buterin 2014] puts all generated event records into a dedicated region of the blockchain state called *event logs*. This region is *write-only* for smart contracts but it can be queried by any external user who runs an Ethereum full node. The original design of the event mechanism in Ethereum is to facilitate the integration of on-chain components and off-chain components in a blockchain-powered application. For example, the transfer function of the popular ERC-20 contract for fungible tokens typically emits a transfer event besides updating the token ledger state in the contract [Fabian Vogelsteller 2015]. It expects that a digital wallet application will run an Ethereum full node as its back-end, monitor these events, and update its front-end GUI accordingly to show the token balance to users.

As smart contracts become more and more complicated and inter-dependent, the existing event logging mechanism in Ethereum becomes increasingly inadequate. In many scenarios, the correctness of one smart contract is now dependent on its timely responses to critical events from other smart contracts. For example, in Ethereum, there are oracle contracts [ChainLink 2020; provable.xyz 2019], which feed off-chain data, such as the digital asset prices, to the blockchain; there are also decentralized finance (DeFi) smart contracts [Foundation 2019; Leshner and Hayes 2019; Oved and

Authors' addresses: Zihan Zhao, University of Toronto, Canada, simon.zhao@mail.utoronto.ca; Sidi Mohamed Beillahi, University of Toronto, Canada, sm.beillahi@utoronto.ca; Ryan Song, University of Toronto, Canada, r.song@mail.utoronto.ca; Yuxi Cai, University of Toronto, Canada, yuxijune.cai@mail.utoronto.ca; Andreas Veneris, University of Toronto, Canada, veneris@eecg.utoronto.ca; Fan Long, University of Toronto, Canada, fanl@cs.toronto.edu.

Mosites 2017], for which the most recent price of a digital asset is critical, e.g., collateral liquidation is required when the asset price drops below a certain threshold. The DeFi contracts therefore have to timely respond to asset price change events from the oracle contracts. Unfortunately, it is impossible to implement such an event-driven execution model in Ethereum. This is because, 1) event logs are write-only for smart contracts and a contract cannot respond to emitted events from other contracts automatically; 2) a smart contract execution can only be triggered (in)directly via function calls by external user transactions.

To this end, many blockchain applications circumvent this problem via *off-chain relay servers* [Denoeud 2020]. A server constantly monitors the blockchain ledger. When a critical event from a source contract (e.g., an oracle contract) occurs, the server will send a *poke transaction* to a target contract (e.g., a DeFi contract) to drive the contract to respond to the event. However, this adhoc solution has two undesirable consequences. First, it creates a central point of failure that defeats the purpose of encoding transaction rules as smart contracts on blockchain platforms. If the off-chain relay server of the target contract fails, the contract will not properly respond to critical events. Secondly, the blockchain platform may not process the poke transaction timely due to insufficient transaction fees or network congestion. The target contract may also undesirably interact with other users before it incorporates the critical updates carried by the poke transaction.

In this paper, we present a novel end-to-end blockchain framework that extends Ethereum to support an event-driven execution model. The core of our framework is SigVM, a novel virtual machine that extends EVM with new opcodes to introduce *signal events*, a special kind of events that enables a new way for multiple contracts to interact with each other. To realize SigVM, we develop SigSolid, a modified version of Solidity programming language that can utilize new opcodes in SigVM, and SigChain, a prototype blockchain that implements SigVM.

In SigVM, contracts can emit signal events, on which other contracts can listen. When a contract listens to a signal event, it binds a function as its handler. When the event is triggered, the handler function will be automatically executed. This new signal event mechanism enables truly decentralized smart contracts to timely respond to critical events, eliminating the reliance on off-chain centralized relay servers.

One challenge SigVM faces is how to integrate the execution of handler functions with the existing smart contract framework. In SigVM, a transaction from a user to a smart contract can emit signal events to trigger multiple handler functions. If we naively implement the execution of these functions synchronously as function calls, the cascading execution process may cause the transaction to exceed the block gas limit. Furthermore, a user who triggers a signal event will have to pay the gas cost for the execution of all of the associated handler functions. This is undesirable, because the user triggering the event is often the service provider while the contracts containing the handler functions are service users. For example, an oracle maintainer sends a transaction to trigger an event to update the price of a digital asset in an oracle contract and this event is listened by many other contracts to react upon the price update. It is counter-intuitive to ask the oracle maintainer to pay for the execution cost of the contracts using the oracle service.

To address this challenge, SigVM executes handler functions asynchronously as *signal transactions*, a special kind of transactions generated by the SigVM execution engine when a signal event is emitted. These special transactions will be packed along with regular transactions in blocks. This asynchronous execution mechanism enables SigVM to circumvent the block gas limit issue. Because SigVM executes signal transactions asynchronously, miners can pack them in separate blocks. This mechanism also enables SigVM to charge the transaction fee of each signal transaction differently to introduce proper incentives. A smart contract that binds a handler function pays the transaction fee of the corresponding signal transaction in SigVM. The transaction fee will be slightly higher than the average of normal transactions in the same block[1], which incentivizes miners to prioritize their execution.

Another challenge SigVM faces is the possibility that a smart contract interacts with other users undesirably before it can integrate critical state updates in an asynchronous signal transactions. Note that the centralized off-chain relay server solution faces the same challenge. The typical solution of paying high transaction fee does not eliminate this risk, because powerful miners can selectively pack *interfering transactions* ahead of the signal/poking transactions.

SigVM addresses this challenge with a novel *contract event lock* mechanism. If a contract has any pending signal transactions, SigVM allows to lock the contract. Any normal transactions interacting with a locked contract become no-ops. The miner who packs the transaction will receive no transaction fees and the transaction will be recycled back to pending transaction pools. It may be packed again in the future when the pending signal transactions are processed and the contract is unlocked. This mechanism effectively stops any interfering transactions from exploiting a contract in the middle of signal event handling.

We evaluate SigVM by reimplementing 23 smart contracts from 13 popular decentralized applications that are critical to the economic ecosystem of Ethereum. Our results show that the signal event mechanism in SigVM efficiently replaces off-chain relay servers and significantly reduces the applications' exposure to interfering transactions. We show that the performance overhead incurred by SigChain over Ethereum blockchain is negligible. Our results show that the SigSolid language powered by SigVM is practical to use. The development effort of migrating smart contracts to SigVM is small, i.e., we modified on average 3.17% of contract code. Using the transactions history for 5 smart contracts, we also show that SigVM gas incentive mechanism is practical. For the majority of the applications, the number of gas units consumed by a signal transaction is smaller than the one consumed by the corresponding poke transaction. This is because of the code simplifications introduced by using SigVM signal events instead of adhoc methods based on off-chain relay servers.

1.1 Contribution

In summary, this paper makes the following contributions:

- **SigVM:** a novel blockchain virtual machine, SigVM, that extends EVM with an event-driven execution model to enable truly decentralized smart contracts (§4-5). Thus, allowing an easy integration into blockchains that work with EVM-based execution environment.
- **Signal transaction and contract event locking:** a novel asynchronous signal transaction design with a formal semantics to address the series of challenges of integrating SigVM with a blockchain platform (§4-5). A novel contract event locking mechanism to prevent interference against signal transactions (§4-5).
- **Implementation:** an end-to-end prototype blockchain platform powered by SigVM (§5). We develop SigSolid, a modified version of Solidity programming language that utilizes the new opcodes in SigVM, and SigChain a prototype blockchain platform that implements SigVM to enable safe, efficient and easy to use event-driven programming model in smart contracts.
- **Experimental evaluation:** an evaluation using 23 smart contracts from 13 popular distributed applications (§6). Our results show that SigVM efficiently eliminates the reliance on relay servers. The development effort of reimplementing existing applications on Ethereum for SigVM is small, i.e., on average 3.17% of the contract code is modified. Compared with Ethereum, SigVM transaction execution runtime overhead is negligible. The results proves that SigVM is practical and enables the development of truly decentralized and robust contracts.

[1]The average value is computed by miners during transactions packing.

Aside from the aforementioned technical sections, we present a motivating example of developing smart contracts in SigVM in Section 2, recall some definitions and terminologies related to blockchain and Ethereum in Section 3, and discuss related work and conclusions in Sections 7 and 8.

```
contract Median {
  uint val; // price
  function peek() public view returns (uint,bool)  //Peek gets current price of digital asset
   { return (val, val > 0); }
+ signal Pr(uint,bool); //price update signal
+ address[] SigT; //Target handler addresses
  //Feed to set current price of a digital asset
  function feed(uint[] p,uint[] v,bytes[] r,bytes[] s, address[] _sigT) public {
   // check the signature info in r, s, v
   ...
   val = computeMedian(p);
+  SigT = _sigT; // targets are only OSM modules
+  Pr.emit(val,val>0).target(SigT).delay(0);//emit to OSM
  } ... }
contract OSM {
  struct Feed { uint val; bool has; }
  Feed cur, nxt; //obtained prices
  uint b; //last price feed block number
+ address[] SigR; //allowed accounts
+ bytes[] SigM; //accessible methods
- function peek() public view returns(uint,bool) //Spotter call peek to acquire price feed
-    { return (cur.val, cur.has); }
+ signal DPr(uint,address) //delayed price update signal
+ address[] SigT; //target handler addresses
- function poke() public {  //Poke to update price from Median
+ function prUpdt(uint wut,bool ok) handler {  //Handler function to receive price from Median
-  require(block.timestamp >= b + HOUR);
-  (uint wut,bool ok) = Median(median).peek();   //peek current price
   if (ok) {
-   cur = nxt;nxt = Feed(wut, true);
-   b = uint(block.timestamp-(block.timestamp % HOUR));
+   DPr.emit(wut,this.address).target(SigT).delay(HOUR);
    } }
  constructor(address median, ...) public {
+  prUpdt.bind(median,Median.Pr,0.1,false,SigR,SigM); //Handler prUpdt binds to signal Pr
   ... } }
- contract Spotter{
-  //poke to update price from OSM
-  function poke(address osm) public {
-   (bytes val, bool has) = OSM(osm).peek();
-   //calculate variable "spot" based on "val"
-   ...
-   vat.file(bytes(osm),"spot",spot);//file price to Vat
-  } ... }
contract Vat {  // move here the body of Spotter contract
+  ...
+ address[] SigR; //allowed accounts
+ bytes[] SigM; //accessible methods
+ function prUpdt(uint data,address osm) handler {  //Handler to receive price from OSM
   //calculate variable "spot" based on "data"
    ...
+  file(bytes(osm),"spot",spot); //file price to Vat
  }
  function grab(...) external {...}  // CDP Confiscation used for CDP liquidation
  function move(...) external {...}  // move function used for Dai join or exit
  constructor(address osm0, address osm1, ...) public {
    // bind handlers to OSMs PriceFeed signal
+   prUpdt.bind(osm0,OSM.DPr,0.1,true,SigR,SigM);
+   prUpdt.bind(osm1,OSM.DPr,0.1,true,SigR,SigM);
    ...  } ... }
```

Listing 1. Simplied code snippet of MakerDAO

2 EXAMPLE

Listing 1 presents the simplified code snippet of parts of MakerDAO to illustrate building autonomous contracts with SigVM. Lines preceded with "-" and "+" are changes we made to the original implementation to adopt SigSolid. MakerDAO is a decentralized finance protocol, which provides collateral-backed stablecoin called Dai [Foundation 2019]. It follows the Maker Protocol to keep Dai softly pegged to USD by a fixed ratio. This is accomplished by backing Dai with crypto assets based on the market prices. In order to generate Dai, users send collaterals to the Collateralized Debt Position smart contract to create a vault. To meet the long-term solvency of the system, the ratio between the deposit to the vault and the Dai the vault owner obtains must be greater than a threshold decided by a MakerDAO governance committee at all time. Once the ratio drops below the set threshold, the Maker Protocol forces the collateral to be liquidated.

MakerDAO components: There are four contracts in Listing 1, `Median`, `OSM`, `Spotter` and `Vat`. A `Median` contract is an oracle that maintains the spot price of one digital asset type. A group of authorized maintainers send batched transactions via the `feed` function to report the real world spot price (line 8). The function computes the median of the reported price as the oracle spot price. The latest oracle price can be accessed via the `peek` function (line 3).

`OSM` is the acronym of Oracle Security Module. `OSM` delays the price obtained from `Median` by an hour. Based on the MakerDAO documentation, this is to allow an emergency committee to supervise the oracle prices. In case of an emergency, the committee may pause the price feed and the MakerDAO system via administrative interfaces (omitted in the code snippet). `Spotter` is a contract that collects spot prices from multiple `OSM` contracts for different kinds of digital asset collaterals.

`Vat` is the core vault engine which stores and tracks all the Dai and collaterals. `Spotter` eventually invokes `Vat` to file the price change (line 43). Note that all other modules rely on `Vat` to implement the desired finance service for users. For example, when a user joins the MakerDAO protocol to mint Dai with Ethereum or other digital assets as collaterals, MakerDAO calls the function `move` (line 55) in `Vat` to update the vault. When a user submits a transaction to liquidate a position (i.e., sell the collaterals), MakerDAO calls the function `grab` (line 54) to update.

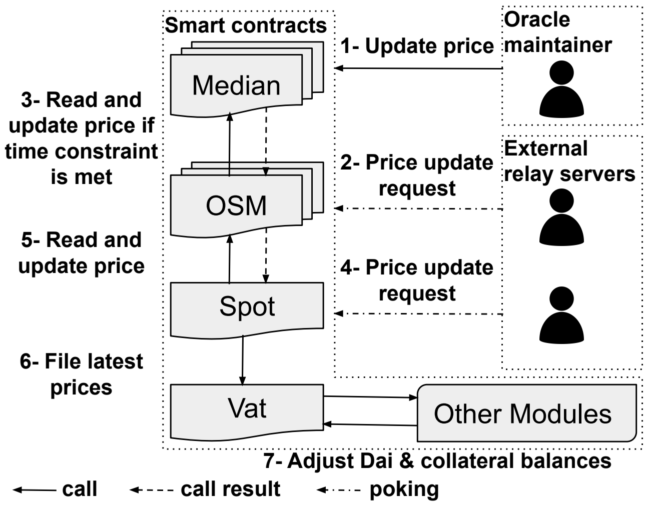

Fig. 1. Overview of smart contracts and user interaction in MakerDAO.

Off-chain relay server poking: It is not feasible to implement this price information propagation fully on chain via function calls because the oracle maintainers and the MakerDAO administrators are different groups of people. The asset price oracle service might be used by many different DeFi contracts and the oracle maintainers are not willing to pay excess transaction fee cost for the executions of MakerDAO. Also, `OSM` needs to introduce one hour delay to the price feed, which is not feasible to implement on-chain in Ethereum.

MakerDAO therefore relies on off-chain relay servers to drive the price information flow. Fig. 1 illustrates the interaction between off-chain servers and the contracts. The off-chain relay servers are expected to call `poke` in `OSM` (line 25) for each digital asset every one hour to extract the price from the `Median` contracts. They are also expected to call `poke` in `Spotter` (line 39) frequently to feed the price information to the core engine. Function `poke` files the price change into `Vat`, which will update its internal state to change the behaviors of the implemented Dai join, exit, and liquidation functionalities accordingly.

Security risks and loss: The off-chain relay server design has significant security risks. Knowing a reasonably accurate price for digital assets is critical for the security of MakerDAO. If the off-chain relay server fails or the network is congested so that poke transactions are not processed in time, the core MakerDAO engine would operate with outdated price information and make incorrect liquidation decisions.

On March 12 2020, the price of crypto currencies dropped significantly. Meanwhile, the Ethereum network became overwhelmed with too many transactions. Critical poke transactions containing price information were delayed, causing the core MakerDAO engine to operate with stale prices for hours. As a result, many MakerDAO users had their positions liquidated while getting much less collateral back comparing to the amount they should have retained with the correct prices. Even after the prices returned to the required level, the collateral was auctioned because its price oracle failed to update price feed. The root cause of this event is the market crash and the network congestion, but the off-chain relay servers design exacerbated the delay during the market collapse. The total financial loss of MakerDAO users during this event is approximately $4.5M [Foundation 2020].

SigSolid powered by SigVM: We next show how to implement the oracle components of MakerDAO in SigSolid, our modified Solidity language that can utilize special SigVM operations for an event-driven execution model.

SigSolid allows a contract to define its signal events via the `signal` keyword (line 5). The contract can emit declared events during its execution together with event data, targeted contracts to receive the events, and a delay in number of blocks (line 13). SigSolid also introduces a new function modifier `handler`. Functions declared with the keyword `handler` are signal handler functions that can be attached to signal events (lines 26 and 49). Once bound, handler functions will be invoked as a separate signal transaction with the data supplied by the event as parameters.

Note that the third parameter of a `bind` statement is a positive real number as the gas incentive provided by the contract. For example, 0.1 in line 35, 58, 59 denotes that the contract is willing to pay a gas price 10% higher than the average gas price of other regular transactions in the same block computed by miners during transactions packing. The fourth parameter of a `bind` statement is either `true` or `false`, indicating whether the generated signal transactions from this handler function will lock the contract from regular transactions or not. If it is `true`, then the contract locking mechanism is enabled. Regular transactions will not be able to interact with the contract if there is a pending signal transaction. The fifth and sixth parameters specify exceptions for the contract locking mechanism, i.e., white-lists of addresses and methods that are allowed for regular transactions during locking (`SigR` and `SigM` at lines 19 and 47).

Implementation of Median, OSM, and Vat in SigSolid: To implement the desired price information flow, we define a signal event in `Median` (line 5) and a signal event in `OSM` (line 23). Instead of passively waiting for other contracts to call `peek`, `Median` emits a price feed `Pr` event whenever there is a valid batch of oracle price updates (line 13). This event is automatically handled by the function in `OSM` at line 26, which in turn emits a delayed price feed event `DPr` with a delay of one hour. The emitted `DPr` will be eventually handled by the `prUpdt` function in `Vat` (line 49) to file the new price information. This function replaces the original `poke` in `Spotter`. The contract `Spotter` is merged into `Vat`.
Event-driven execution with SigVM: Once deployed on SigChain, the modified code in Listing 1 will enable the desirable information to flow fully on-chain. When the oracle maintainers send a transaction to invoke the function `feed` in `Median` (line 8) to provide a new price. The contract will emit the `Pr` event (line 15) with the calculated mean asset price as the event data. Because `OSM` registered the function `prUpdt` (line 26) as the handler for `Pr` event, SigVM will generate a signal transaction to invoke `prUpdt` with the corresponding event data as the parameter. SigVM will execute the generated signal transaction automatically and asynchronously. The transaction will cascadingly emit the `DPr` signal event (lines 32) with the asset price and an address id (which represents the digital asset type). This event will be emitted with a delay of one hour. Because `Vat` registers the function `prUpdt` to handle the `DPr` event from `OSM` contracts, after one hour delay, SigVM will generate and execute a signal transaction to invoke `prUpdt` with the asset price and the asset address as parameters automatically. `prUpdt` in `Vat` will finally file the price change at line 43.
Contract event locking: The advantage of setting the handler function in `Vat` is to utilize the contract event lock mechanism in SigVM. After the one-hour delay of the `DPr` event, SigVM will lock the `Vat` contract from arbitrary regular transactions until the signal transaction that invokes `prUpdt` is executed. This prevents `Vat` from interacting with potentially malicious users before it incorporates critical updates from the signal transaction, e.g. minting Dai or liquidating collaterals with outdated digital asset prices. This lock mechanism is lightweight because it only affects one contract. Other contracts deployed on the blockchain are not affected.
Transaction fees: Different from the Ethereum gas mechanism that transaction fees are always paid by external users. The transactions fees of signal transactions in SigChain are paid by the smart contracts who register the corresponding handler functions. In our example, `OSM` and `Vat` must have sufficient native token balance to cover the signal transaction fees. We believe this is a much smaller burden than maintaining an off-chain relay server to send poke transactions.
Advantages: This example highlights the advantages of SigVM. SigVM eliminates the dependency on the unreliable poking mechanism and off-chain relay servers. The execution model of SigVM guarantees that the core vault engine operates with the one-hour delayed prices reported by the oracle maintainers. Listing 1 highlights the expressiveness of SigSolid. Challenging features like time delays can be implemented in SigSolid in a straightforward way.

3 BACKGROUND

In this section, we briefly recall some background on the Ethereum blockchain which is also valid for other blockchains, e.g., Avalanche [Sekniqi et al. 2020], Conflux [Li et al. 2020b], and Near [Foundation 2021], that work with EVM-based execution environment.

3.1 Ethereum blockchain

The Ethereum blockchain is a state machine constituted of a global state and transactions that modify the state. The state includes account information such as the account Ether balance. Ethereum supports two types of accounts: user accounts and smart contract accounts. Each account is associated with a unique address. Users can interact with the blockchain by issuing transactions using their user accounts. Smart contract accounts are software objects that manage transactions. Beside the account balance, a smart contract state stores the account's storage trie and the EVM code that executes incoming transactions. Smart contracts can create internal transactions to other smart contracts. The internal transactions are nested from a top-level transaction, an external transaction, initiated by a user account. If a transaction is aborted then the effects of all nested internal transactions will be reversed.

3.2 Transactions Execution and Gas

A transaction contains the addresses of sender and recipient accounts, the transferred value of native tokens (can be zero), a data, and a gas value. If a transaction recipient account is a user account then the transaction's data is empty and the transaction constitutes of transferring the value of native tokens from the sender account to the recipient account. Otherwise, if the transaction recipient account is a smart contract then the transaction's data identifies a function of the recipient smart contract's code together with arguments passed to the function. When the transaction is received, the corresponding smart contract's function is executed by the EVM, modifying the smart contract's storage trie accordingly. The execution of each EVM command, such as read/write operations on the underlying storage trie, is associated with a gas fee. The transaction's gas value must exceed the accumulated gas fees at the end of execution, otherwise the transaction is aborted.

3.3 Events

Important components of smart contracts are events. Events in EVM allow to log meaningful changes to a smart contact that the smart contract wants to communicate to external entities connected to the blockchain, e.g., DApps, so they listen to these events and act accordingly. For instance, as shown in Listing 2 when an address transfer an ERC20 token to another address, the ERC 20 token smart contract will emit a *Transfer* event containing all of the data about the token transfer (line 11). The logs of emitted events are stored on the blockchain and are accessible through the address of the smart contract account that emitted them. EVM does not yet support an event-driven programming model, i.e., smart contracts can emit events but cannot listen to them.

```
1  contract ERC20 {
2      mapping(address => uint256) balances;
3      mapping(address => mapping (address => uint256)) allowed;
4  
5      event Transfer(address from, address  to, uint256 value);
6  
7      function transfer(address receiver, uint256 numTokens) public returns (bool) {
8          require(numTokens <= balances[msg.sender]);
9          balances[msg.sender] = balances[msg.sender]-numTokens;
10         balances[receiver] = balances[receiver]+numTokens;
11         emit Transfer(msg.sender, receiver, numTokens);
12         return true;
13     }
14 }
```

Listing 2. Code snippet of ERC 20 token smart contract.

4 SIGVM DESIGN

This section formalizes the design of SigVM. Similar to other smart contract virtual machines like EVM [Buterin 2014], SigVM contains two layers, the virtual machine execution layer that dictates how SigVM executes a transaction and the block processing layer that dictates how the blockchain state evolves over multiple transactions in a block.

$\langle prog \rangle$ ::= `program` $\langle inst \rangle^*$

$\langle inst \rangle$ ::= $\langle EVMinst \rangle \mid \langle SIGinst \rangle$

$\langle SIGinst \rangle$ ::= `createsignal` sn | `deletesignal` sn | `detach` (m, s) | `bind` $(m, \mathsf{s}, \mathsf{gr}, \mathsf{blk}, \mathsf{s\text{-}r}, \mathsf{s\text{-}m})$ | `emit` $(\mathsf{s}, \mathsf{s\text{-}t}, \mathsf{d})$

Fig. 2. Core language signal related instructions syntax of SigVM. a^* indicates zero or more occurrences of a.

4.1 Core SigVM Language

Fig. 2 lists the syntax of a simple programming language used to formalize our approach. Our language extends the standard EVM operations `EVMinst` (e.g., `load` and `push`) with new operations `SIGinst`, e.g., `createsignal` and `bind`, to enable an event-driven execution model under SigVM. For brevity, in Fig. 2 we omit standard EVM operations `EVMinst` since they are not necessary in understanding the design of SigVM. However, our implementation of SigVM actually supports all standard EVM opcodes including arithmetic and inter-contract call operations.

In `SIGinst`, `createsignal` and `deletesignal` opcodes handle the creation and deletion of a signal event with name sn, respectively. The opcodes `bind` and `detach` allow handler functions to listen and unlisten to signal events.

The function `bind`$(m, \mathsf{s}, \mathsf{gr}, \mathsf{blk}, \mathsf{s\text{-}r}, \mathsf{s\text{-}m})$ attaches a handler function m to a signal event $\mathsf{s} \in \mathbb{S}$ with a gas ratio gr. The signal transaction fee is computed by miners by calculating the average gas price of regular transactions packed in the same block multiplied by $1 + \mathsf{gr}$. The last three parameters of `bind` dictates how the locking mechanism is enforced for the signal event: a boolean flag blk which when it is false regular transactions are allowed to execute. Otherwise, when blk is true only regular transactions initiated by accounts with addresses in the array s-r and are executing functions in the array s-m are allowed to execute. The contract is locked for other regular transactions until the pending signal transactions finish. `detach`(m, s) detaches a handler function m from a signal s. Finally, `emit`$(\mathsf{s}, \mathsf{s\text{-}t}, \mathsf{d})$ emits signal events for signal s with a delay of d. s-t is an array of addresses that specifies the contracts that will be poked by the emitted signal. If s-t is not empty then s pokes only contracts with addresses in s-t. Otherwise, when s-t is empty, s pokes all contracts that contain handler functions attached to s. For each poked contract, a signal transaction that executes the corresponding handler function is created.

4.2 Operational Semantics

A program configuration in SigVM is a tuple $\mu = (\mathsf{gas}, \mathsf{gcf}, \mathsf{stack}, \mathsf{bal}, \mathsf{s\text{-}h}, \mathsf{e\text{-}sig})$ where *gas* is the gas counter, gcf is composed of the persistent valuation of program variables, stack is the call stack, bal is a mapping from addresses to the corresponding balances, s-h is a mapping that maps each signal to a set of handlers that are attached to this signal, and e-sig is a set of emitted signal transactions.

The activation frames in stack are represented using tuples $(i, m, \mathsf{ad}, \ell)$ where $i \in \mathbb{T}$ is a task (invocation) identifier, $m \in \mathbb{M}$ is a function name, $\mathsf{ad} \in \mathbb{A}$ is the address of the contract that m belongs to, and ℓ is a valuation of local variables, including a program counter.

A signal identifier is a tuple $\mathsf{s} = (\mathsf{ad}, \mathsf{sn})$, where $\mathsf{ad} \in \mathbb{A}$ is the address of the contract emitting the signal and sn is the signal name. $\mathsf{s\text{-}h}[\mathsf{s}]$ is a set of tuples $(m, \mathsf{ad}, \mathsf{gr}, \mathsf{blk}, \mathsf{s\text{-}r}, \mathsf{s\text{-}m})$ where m is the name of the handler function bound to the signal s by the contract with the address ad, and gr is the gas ratio. The parameters blk, s-r, and s-m dictate the locking mechanism. blk is a boolean, s-r is an array of addresses, and s-m is an array of functions signatures. We assume that a signal $\mathsf{s} = (\mathsf{ad}, \mathsf{sn})$ exists only if $\mathsf{s\text{-}h}[\mathsf{s}] \neq \mathsf{nil}$ and if a signal is not attached to any method handler then we have $\mathsf{s\text{-}h}[\mathsf{s}] = \emptyset$.

The activation frames in the emitted signal transactions set e-sig are represented using tuples $(j, \mathsf{s}, m, \mathsf{ad}, \mathsf{gr}, \mathsf{d})$, where $j \in \mathbb{SI}$ is a unique identifier of the signal transaction, $\mathsf{s} = (\mathsf{ad}', \mathsf{sn})$ is the signal identifier, m is the handler method name, ad is the address of the contract containing the handler method, gr is the gas ratio, and d is the delay.

$$\frac{\begin{array}{c}\texttt{createsignal}\ \text{sn} \in \text{inst}(\ell(\text{pc})) \qquad \text{s-h}[(\text{ad},\text{sn})] = \text{nil} \\ \text{g} := \text{gascost}(\texttt{createsignal}) \qquad \text{s-h}' := \text{s-h}[(\text{ad},\text{sn}) \mapsto \emptyset] \qquad \ell' := \ell[\text{pc} \mapsto \text{next}(\ell(\text{pc}))]\end{array}}{(\text{gas}, \_, (i, m, \text{ad}, \ell) \circ \text{stack}, \text{bal}, \text{s-h}, \_) =[\![\ \texttt{createsignal}\ \text{sn}\ ]\!]\Rightarrow (\text{gas} + \text{g}, \_, (i, m, \text{ad}, \ell') \circ \text{stack}, \text{bal}, \text{s-h}', \_)}$$

$$\frac{\begin{array}{c}\texttt{deletesignal}\ \text{sn} \in \text{inst}(\ell(\text{pc})) \qquad \text{s-h}[(\text{ad},\text{sn})] \neq \text{nil} \\ \text{g} := \text{gascost}(\texttt{deletesignal}) \qquad \text{s-h}' := \text{s-h}[(\text{ad},\text{sn}) \mapsto \text{nil}] \qquad \ell' := \ell[\text{pc} \mapsto \text{next}(\ell(\text{pc}))]\end{array}}{(\text{gas}, \_, (i, m, \text{ad}, \ell) \circ \text{stack}, \text{bal}, \text{s-h}, \_) =[\![\ \texttt{deletesignal}\ \text{sn}\ ]\!]\Rightarrow (\text{gas} + \text{g}, \_, (i, m, \text{ad}, \ell') \circ \text{stack}, \text{bal}, \text{s-h}', \_)}$$

$$\frac{\begin{array}{c}\texttt{bind}\ (m', \text{s}, \text{gr}, \text{blk}, \text{s-r}, \text{s-m}) \in \text{inst}(\ell(\text{pc})) \qquad \text{s} = (\text{ad}', \text{sn}) \qquad \text{s-h}[\text{s}] \neq \text{nil} \qquad (\_, \text{ad}, \_, \_, \_) \notin \text{s-h}[\text{s}] \\ \text{g} := \text{gascost}(\texttt{bind}) \qquad \text{s-h}' := \text{s-h}[\text{s} \mapsto (m', \text{ad}, \text{gr}, \text{blk}, \text{s-r}, \text{s-m}) \uplus \text{s-h}[\text{s}]] \qquad \ell' := \ell[\text{pc} \mapsto \text{next}(\ell(\text{pc}))]\end{array}}{(\text{gas}, \_, (i, m, \text{ad}, \ell) \circ \text{stack}, \text{bal}, \text{s-h}, \_) =[\![\ \texttt{bind}\ (m', \text{s}, \text{gr}, \text{blk}, \text{s-r}, \text{s-m})\ ]\!]\Rightarrow (\text{gas} + \text{g}, \_, (i, m, \text{ad}, \ell') \circ \text{stack}, \text{bal}, \text{s-h}', \_)}$$

$$\frac{\begin{array}{c}\texttt{detach}\ (m', \text{s}) \in \text{inst}(\ell(\text{pc})) \qquad \text{s} = (\text{ad}', \text{sn}) \qquad (m', \text{ad}, \_, \_, \_) \in \text{s-h}[\text{s}] \\ \text{g} := \text{gascost}(\texttt{detach}) \qquad \text{s-h}' := \text{s-h}[\text{s} \mapsto \text{s-h}[\text{s}] \setminus (m', \text{ad}, \_, \_, \_)] \qquad \ell' := \ell[\text{pc} \mapsto \text{next}(\ell(\text{pc}))]\end{array}}{(\text{gas}, \_, (i, m, \text{ad}, \ell) \circ \text{stack}, \text{bal}, \text{s-h}, \_) =[\![\ \texttt{detach}\ (m', \text{s})\ ]\!]\Rightarrow (\text{gas} + \text{g}, \_, (i, m, \text{ad}, \ell') \circ \text{stack}, \text{bal}, \text{s-h}', \_)}$$

$$\frac{\begin{array}{c}\texttt{emit}\ (\text{s}, \text{s-t}, \text{d}) \in \text{inst}(\ell(\text{pc})) \\ \text{g} := \text{gascost}(\texttt{emit}) \qquad \text{e-sig}' := \text{emitSig}(\text{s}, \text{e-sig}, \text{s-h}, \text{s-t}, \text{d}) \qquad \ell' := \ell[\text{pc} \mapsto \text{next}(\ell(\text{pc}))]\end{array}}{(\text{gas}, \_, (i, m, \text{ad}, \ell) \circ \text{stack}, \text{bal}, \text{s-h}, \text{e-sig}) =[\![\ \texttt{emit}\ (\text{s}, \text{s-t}, \text{d})\ ]\!]\Rightarrow (\text{gas} + \text{g}, \_, (i, m, \text{ad}, \ell') \circ \text{stack}, \text{bal}, \text{s-h}, \text{e-sig}')}$$

Fig. 3. Program semantics in SigVM. For a function f, we use $f[a \mapsto b]$ to denote a function g such that $g(c) = f(c)$ for all $c \neq a$ and $g(a) = b$. The function inst returns the instruction at some given control location while next gives the next instruction to execute. gascost returns the gas price for a given opcode. We use $\circ$ to denote sequence concatenation.

Fig. 3 presents the small step semantics for SigVM signal related opcodes. The notation $\mu =[\![\ a\ ]\!]\Rightarrow \mu'$ represents the state transition of μ to μ' after executing operation a.

A transition labeled by `createsignal` sn corresponds to the creation of a new signal with the name sn. The new signal is identified with s = (ad, sn) where is ad is the address of the current contract. s-h[s] is set to the empty set. The gas counter is incremented with the gas cost of the operation. A transition labeled by `deletesignal` sn corresponds to the removal of the signal with the name sn. The signal is identified with s = (ad, sn) where ad is the address of the current contract. s-h[s] is set to nil.

A transition labeled by `bind` $(m', \text{s}, \text{gr}, \text{blk}, \text{s-r}, \text{s-m})$ corresponds to attaching the signal s to the method m' and assigning gr as the gas ratio with locking parameters blk, s-r, and s-m. s must exist, i.e., s-h[s] $\neq$ nil. Also, the current contract must not have bound another method to s, i.e., $(\_, \text{ad}, \_, \_, \_) \notin \text{s-h}[\text{s}]$ where ad is the contract address. We then insert $(m', \text{ad}, \text{gr}, \text{blk}, \text{s-r}, \text{s-m})$ in the set s-h[s] to mark that the contract ad assigned the method m' to s with a gas ratio gr. A transition labeled by `detach` (m', s) corresponds to detaching a handler m' from a signal s. The transition results in the removal of $(m', \text{ad}, \_, \_, \_)$ from the set s-h[s].

Finally, a transition labeled by `emit` (s, s-t, d) corresponds to emitting a signal s after a delay d. If s-t is not empty s pokes contracts associated with addresses in s-t. Otherwise, s pokes all contracts with attached handlers for s. We use the function emitSig defined in Algorithm 1 to update the emitted signal transactions set e-sig with the newly emitted signal transactions accordingly.

An execution of an externally invoked method m of a contract stored in the address ad is a sequence $\rho = \mu_0 =[\![\ a_1\ ]\!]\Rightarrow \mu_1 =[\![\ a_2\ ]\!]\Rightarrow \ldots =[\![\ a_n\ ]\!]\Rightarrow \mu$ of transitions starting in the configuration $\mu_0 = (0, \text{gcf}, (i, m, \text{ad}, \ell_0), \text{bal}, \text{s-h}, \epsilon)$ where $\ell_0 = \text{locInit}(\text{gcf}, m)$ represents the initial state of m, and leading to a configuration $\mu = (\text{gas}, \text{gcf}', \epsilon, \text{bal}', \text{s-h}', \text{e-sig})$ where the call stack is empty. We use the notation $\mu = \rho_m(\mu_0)$ to say that the execution ρ of m transforms μ_0 to μ.

Note that for brevity we omit the rules that check the current used gas against the limit to terminate execution early. In particular, during transaction execution each time an instruction

Algorithm 1 Signals to emit.

```
1: procedure emitSig(s, e-sig, s-h, s-t, d)
2:     $Q \leftarrow$ e-sig;
3:     output $Q$;
4:     for each $(m, \mathrm{ad}, \mathrm{gr}, \_, \_, \_) \in \mathrm{s\text{-}h}[\mathrm{s}]$
5:         if $\mathrm{s\text{-}t} = \epsilon$ or $\mathrm{ad} \in \mathrm{s\text{-}t}$
6:             $Q \leftarrow (j, \mathrm{s}, m, \mathrm{ad}, \mathrm{gr}, \mathrm{d}) \uplus Q$;    $j \in \mathbb{SI}$ $fresh$
7: end procedure
```

opcode operation is executed, its corresponding gas consumption is recorded and the total used gas is calculated. The execution will be terminated early if the total used gas is greater than the pre-defined gas limit. This is the standard practice in Ethereum to address the termination problem and our SigVM implementation follows the same principle.

4.3 Transaction Execution

Blockchain state: A blockchain state in SigVM is a tuple $\sigma = (\mathrm{h}, \mathrm{bcf}, \mathrm{s\text{-}h}, \mathrm{eh\text{-}sig})$ where h is the current block height, bcf maps each account address ad to $\mathrm{bcf(ad)} = (\mathrm{non}, \mathrm{b}, \mathrm{gcf}, \mathrm{c})$ where non is the account nonce, b is the account balance, gcf is the persistent valuation of contract variables, and c is the immutable contract code. eh-sig is a mapping that maps each block height to a set of emitted signal transactions pending until the block height is attained to be executed. For a block height h, the activation frames in the emitted signal transactions set eh-sig[h] are represented using tuples $(j, \mathrm{s}, m, \mathrm{ad}, \mathrm{gr})$, where j is the unique signal transaction identifier, s is the signal identifier, m is the handler method name, ad is the address of the contract containing the handler method, and gr is the gas ratio.

Transactions execution: Fig. 4 presents the transaction execution rules for SigVM. We use $\sigma =[\![\text{ tx }]\!]\Rightarrow^t \sigma'$ to denote the blockchain state transition from σ to σ' after executing a transaction tx. In SigVM, we note three kinds of transactions: (1) create transaction which creates a new account and transfers balance, (2) regular transaction which executes a smart contract method and transfers balance, and (3) signal transaction which is a special transaction generated by signal emitted in previous blocks and does not transfer balance.

A transition labeled by createTx $(\mathrm{ad}_s, \mathrm{non}_s, \mathrm{b}_0, m, \mathrm{c}, \mathrm{gp})$ corresponds to an account creation transaction initiated by the account with the address ad_s and nonce non_s transferring a balance b_0 to the new contract account. c parameter is for the code of a smart contract to deploy in the new account and m is the name of constructor method in the contract. gp is the gas price payed by the sender to execute the transaction. The transition creates a fresh address ad_n for the new account and ensures that the balance of the sender is sufficient to pay for the transaction. Also, the transition includes an execution of the constructor method m, i.e., ρ_m, over the newly created contract initial state. The emitted signal transactions map eh-sig is updated with the newly emitted signals during the execution of m, i.e., e-sig using the function sigPartition that partitions emitted signal transactions based on the values of the block height h and the delay period d defined in Algorithm 2.

A transition labeled by regularTx $(\mathrm{ad}_s, \mathrm{ad}_t, m, \mathrm{val}, \mathrm{gp})$ corresponds to a regular transaction initiated by the account with the address ad_s targeting the method m in the contract associated with the address ad_t and transferring a balance val to ad_t. gp is the gas price payed by the sender to execute the transaction. The transition ensures that the balance of the sender is sufficient to pay for the transaction. To ensure that the regular transaction is not executing a locked smart contract because of pending signal transactions we define a boolean function lockPerm which returns true

$$\dfrac{\begin{array}{c}(\mathsf{non}_s, \mathsf{b}_s, \mathsf{gcf}_s, \mathsf{c}_s) = \mathsf{bcf}(\mathsf{ad}_s) \qquad \mathsf{b}_s \geq \mathsf{gp} + \mathsf{b}_0 \qquad \mathsf{ad}_n \in \mathbb{A}\ \mathit{fresh} \\ \forall\ \mathsf{ad} \in \mathbb{A}.\ \mathsf{bcf}[\mathsf{ad}] = (\_, \mathsf{b}, \_, \_) \implies \mathsf{bal}_1[\mathsf{ad}] := \mathsf{b} \qquad \mathsf{bal}_2 := \mathsf{bal}_1[\mathsf{ad}_s \mapsto \mathsf{bal}_1[\mathsf{ad}_s] - \mathsf{b}_0;\ \mathsf{ad}_n \mapsto \mathsf{bal}_1[\mathsf{ad}_n] + \mathsf{b}_0] \\ \mathsf{gcf}_0 := \mathsf{init}(\mathsf{ad}_n) \qquad \ell_0 := \mathsf{locInit}(\mathsf{gcf}_0, m) \qquad (\mathsf{gas}, \mathsf{gcf}, \epsilon, \mathsf{bal}, \mathsf{s\text{-}h}', \mathsf{e\text{-}sig}) := \rho_m(0, \mathsf{gcf}_0, (i, m, \mathsf{ad}_n, \ell_0), \mathsf{bal}_2, \mathsf{s\text{-}h}, \epsilon) \\ \mathsf{gp} \geq \mathsf{gas} \times \mathsf{unitGasPrice}(\mathsf{h}) \qquad \mathsf{eh\text{-}sig}' := \mathsf{sigPartition}(\mathsf{h}, \mathsf{eh\text{-}sig}, \mathsf{e\text{-}sig}) \qquad \mathsf{bcf}_1 := \mathsf{updBal}(\mathsf{bcf}, \mathsf{bal}) \\ \mathsf{bcf}_2 := \mathsf{bcf}_1[\mathsf{ad}_s \mapsto (\mathsf{non}_s + 1, \mathsf{bal}[\mathsf{ad}_s] - \mathsf{gp}, \mathsf{gcf}_s, \mathsf{c}_s);\ \mathsf{ad}_n \mapsto (\mathsf{non}_s, \mathsf{bal}[\mathsf{ad}_n], \mathsf{gcf}, \mathsf{c})]\end{array}}{(\mathsf{h}, \mathsf{bcf}, \mathsf{s\text{-}h}, \mathsf{eh\text{-}sig}) = [\![\ \mathtt{createTx}\ (\mathsf{ad}_s, \mathsf{non}_s, \mathsf{b}_0, m, \mathsf{c}, \mathsf{gp})\ ]\!]\Rightarrow^t (\mathsf{h}, \mathsf{bcf}_2, \mathsf{s\text{-}h}', \mathsf{eh\text{-}sig}')}$$

$$\dfrac{\begin{array}{c}(\mathsf{non}_s, \mathsf{b}_s, \mathsf{gcf}_s, \mathsf{c}_s) = \mathsf{bcf}(\mathsf{ad}_s) \qquad (\mathsf{non}_t, \mathsf{b}_t, \mathsf{gcf}_t, \mathsf{c}_t) = \mathsf{bcf}(\mathsf{ad}_t) \\ \forall\ \mathsf{ad} \in \mathbb{A}.\ \mathsf{bcf}[\mathsf{ad}] = (\_, \mathsf{b}, \_, \_) \implies \mathsf{bal}_1[\mathsf{ad}] := \mathsf{b} \qquad \mathsf{bal}_2 := \mathsf{bal}_1[\mathsf{ad}_s \mapsto \mathsf{bal}_1[\mathsf{ad}_s] - \mathsf{val};\ \mathsf{ad}_t \mapsto \mathsf{bal}_1[\mathsf{ad}_t] + \mathsf{val}] \\ \mathsf{b}_s \geq \mathsf{gp} + \mathsf{val} \qquad \mathsf{lockPerm}(\mathsf{h}, \mathsf{s\text{-}h}, \mathsf{eh\text{-}sig}, \mathsf{ad}_t, m, \mathsf{ad}_s) \\ \ell_0 := \mathsf{locInit}(\mathsf{gcf}_t, m) \qquad (\mathsf{gas}, \mathsf{gcf}, \epsilon, \mathsf{bal}, \mathsf{s\text{-}h}', \mathsf{e\text{-}sig}) := \rho_m(0, \mathsf{gcf}_t, (i, m, \mathsf{ad}_t, \ell_0), \mathsf{bal}_2, \mathsf{s\text{-}h}, \epsilon) \\ \mathsf{gp} \geq \mathsf{gas} \times \mathsf{unitGasPrice}(\mathsf{h}) \qquad \mathsf{eh\text{-}sig}' := \mathsf{sigPartition}(\mathsf{h}, \mathsf{eh\text{-}sig}, \mathsf{e\text{-}sig}) \qquad \mathsf{bcf}_1 := \mathsf{updBal}(\mathsf{bcf}, \mathsf{bal}) \\ \mathsf{bcf}_2 := \mathsf{bcf}_1[\mathsf{ad}_s \mapsto (\mathsf{non}_s + 1, \mathsf{bal}[\mathsf{ad}_s] - \mathsf{gp}, \mathsf{gcf}_s, \mathsf{c}_s);\ \mathsf{ad}_t \mapsto (\mathsf{non}_t, \mathsf{bal}[\mathsf{ad}_t], \mathsf{gcf}, \mathsf{c}_t)]\end{array}}{(\mathsf{h}, \mathsf{bcf}, \mathsf{s\text{-}h}, \mathsf{eh\text{-}sig}) = [\![\ \mathtt{regularTx}\ (\mathsf{ad}_s, \mathsf{ad}_t, m, \mathsf{val}, \mathsf{gp})\ ]\!]\Rightarrow^t (\mathsf{h}, \mathsf{bcf}_2, \mathsf{s\text{-}h}', \mathsf{eh\text{-}sig}')}$$

$$\dfrac{\begin{array}{c}\exists\ \mathsf{h}' \leq \mathsf{h}.\ \mathsf{eh\text{-}sig}[\mathsf{h}'] = (j, \mathsf{s}, m, \mathsf{ad}_t, \mathsf{g}) \uplus \mathsf{q} \qquad (\mathsf{non}_t, \mathsf{b}_t, \mathsf{gcf}_t, \mathsf{c}_t) = \mathsf{bcf}(\mathsf{ad}_t) \\ \mathsf{gp} := \mathsf{gas} \times \mathsf{unitSigGasPrice}(\mathsf{h}, \mathsf{gr}) \qquad \mathsf{b}_t \geq \mathsf{gp} \qquad \forall\ \mathsf{ad} \in \mathbb{A}.\ \mathsf{bcf}[\mathsf{ad}] = (\_, \mathsf{b}, \_, \_) \implies \mathsf{bal}_1[\mathsf{ad}] := \mathsf{b} \\ \ell_0 := \mathsf{locInit}(\mathsf{gcf}_t, m) \qquad (\mathsf{gas}, \mathsf{gcf}, \epsilon, \mathsf{bal}, \mathsf{s\text{-}h}', \mathsf{e\text{-}sig}) := \rho_m(0, \mathsf{gcf}_t, (i, m, \mathsf{ad}_t, \ell_0), \mathsf{bal}_1, \mathsf{s\text{-}h}, \epsilon) \\ \mathsf{eh\text{-}sig}_1 := \mathsf{eh\text{-}sig}[\mathsf{h}' \mapsto \mathsf{q}] \qquad \mathsf{eh\text{-}sig}_2 := \mathsf{sigPartition}(\mathsf{h}, \mathsf{eh\text{-}sig}_1, \mathsf{e\text{-}sig}) \\ \mathsf{bcf}_1 := \mathsf{updBal}(\mathsf{bcf}, \mathsf{bal}) \qquad \mathsf{bcf}_2 := \mathsf{bcf}_1[\mathsf{ad}_t \mapsto (\mathsf{non}_t + 1, \mathsf{bal}[\mathsf{ad}_t] - \mathsf{gp}, \mathsf{gcf}, \mathsf{c}_t)]\end{array}}{(\mathsf{h}, \mathsf{bcf}, \mathsf{s\text{-}h}, \mathsf{eh\text{-}sig}) = [\![\ \mathtt{signalTx}\ (j, \mathsf{s}, m, \mathsf{ad}_t, \mathsf{gr})\ ]\!]\Rightarrow^t (\mathsf{h}, \mathsf{bcf}_2, \mathsf{s\text{-}h}', \mathsf{eh\text{-}sig}_2)}$$

Fig. 4. Transactions execution rules. init returns the initial state of a contract persistent variables. unitGasPrice returns the unit gas price at the block h. unitSigGasPrice returns the average unit gas price payed by regular transactions packed in the block h multiplied by 1 + gr. updBal(bcf, bal) returns bcf_1 s.t. for every bcf[ad] = (non, b, gcf, c), $\mathsf{bcf}_1[\mathsf{ad}]$ = (non, bal[ad], gcf, c).

Algorithm 2 Signal transactions partition.

```
1: procedure sigPartition(h, eh-sig, e-sig)
2:     $Q \leftarrow$ eh-sig;
3:     output $Q$;
4:     for each $(j, \mathsf{s}, m, \mathsf{ad}, \mathsf{gr}, \mathsf{d}) \in$ e-sig
5:         $\mathsf{h}' \leftarrow \mathsf{h} + \mathsf{d}$;
6:         $Q \leftarrow Q[\mathsf{h}' \mapsto (j, \mathsf{s}, m, \mathsf{ad}, \mathsf{gr}) \uplus Q[\mathsf{h}']]$;
7: end procedure
```

when the regular transaction is allowed to execute and false otherwise.

$$\begin{aligned}\mathsf{lockPerm}(\mathsf{h}, \mathsf{s\text{-}h}, \mathsf{eh\text{-}sig}, \mathsf{ad}_t, m, \mathsf{ad}_s) = \forall\ \mathsf{s} \in \mathbb{S}.\ (\ &(\forall\ \mathsf{h}' \leq \mathsf{h}.\ \nexists\ (\_, \mathsf{s}, \_, \mathsf{ad}_t, \_, \_) \in \mathsf{eh\text{-}sig}[\mathsf{h}']) \lor \\ &(\exists!\ (\_, \mathsf{ad}_t, \_, \mathsf{blk}, \mathsf{s\text{-}r}, \mathsf{s\text{-}m}) \in \mathsf{s\text{-}h}[\mathsf{s}].\ \neg\mathsf{blk} \lor (\mathsf{ad}_s \in \mathsf{s\text{-}r} \land m \in \mathsf{s\text{-}m}))\ )\end{aligned}$$

∃! denotes unique existential quantifier. lockPerm returns true when there are no pending signal transactions ready to be executed and are associated with the contract ad_t that is targeted by the current regular transaction or for any pending signal transaction that can be executed we have that regular transactions initiated by ad_s and are calling the method m are allowed to execute. If a contract detaches a handler from a signal while signal transactions are queued, regular transactions will be blocked until the signal transactions are executed (lockPermission returns false).

A transition labeled by `signalTx` $(j, \mathsf{s}, m, \mathsf{ad}_t, \mathsf{gr})$ corresponds to a signal transaction executing a method m in the contract with the address ad_t. gr is the gas ratio. Similar to above, the transition ensures that the balance of ad_t is sufficient to pay for the transaction. We remove the executed signal transaction from the set of pending signal transactions at a block height less or equal to the current block height, i.e., $\exists\ \mathsf{h}' \leq \mathsf{h}\ (j, \mathsf{s}, m, \mathsf{ad}_t, \mathsf{gr}) \in \mathsf{eh\text{-}sig}[\mathsf{h}']$.

Block execution: an execution of a new block of transactions blc in SigVM is a sequence $\sigma_0 =⟦ \mathsf{inc} ⟧\Rightarrow^{h+1} \sigma_1 ::= \mathsf{inc}(\sigma_0) =⟦ \mathsf{tx}_1 ⟧\Rightarrow^{t} \sigma_2 =⟦ \mathsf{tx}_2 ⟧\Rightarrow^{t} \ldots =⟦ \mathsf{tx}_{n-1} ⟧\Rightarrow^{t} \sigma_n$ of transitions starting in the initial blockchain state $\sigma_0 = (\mathsf{h}, \mathsf{bcf}_0, \mathsf{s\text{-}h}_0, \mathsf{eh\text{-}sig}_0)$ and leading to a blockchain state $\sigma_n = (\mathsf{h}+1, \mathsf{bcf}_n, \mathsf{s\text{-}h}_n, \mathsf{eh\text{-}sig}_n)$ where the transition $\sigma_0 =⟦ \mathsf{inc} ⟧\Rightarrow^{h+1} \sigma_1 ::= \mathsf{inc}(\sigma_0)$ simply increments the block height in the blockchain state σ_0 resulting in $\mathsf{inc}(\sigma_0) = (\mathsf{h}+1, \mathsf{bcf}_0, \mathsf{s\text{-}h}_0, \mathsf{eh\text{-}sig}_0)$ and tx_1, ..., tx_{n-1} are the transitions of the transactions constituting the block blc, i.e., blc = $\{\mathsf{tx}_1 \ldots \mathsf{tx}_{n-1}\}$.

5 IMPLEMENTATION

In this section we describe an implementation of an end-to-end blockchain platform that supports SigVM. Our implementation consists of two main components: SigSolid, an extension of Solidity language together with a compiler to support SigVM features, and SigChain, a blockchain client based on the OpenEthereum client [Technologies 2019]. As input, our implementation accepts smart contracts written in SigSolid and it then deploys and executes them using SigChain. Our implementation supports all EVM opcodes with the addition of SigVM opcodes for signal events presented in Section 4.

5.1 SigSolid

Our SigSolid language provides high-level language features for using SigVM opcodes to create, bind, and emit signal events while maintaining support for all Solidity language features. The SigSolid compiler extends the Solidity compiler [Ethereum 2020] with a new preprocessing parser that transforms signal related statements to inlined assembly instructions that use the corresponding SigVM opcodes. The resulting code is then complied to assembly code using the Solidity compiler which bypasses the inlined assembly code. The outputted byte code is produced from the entire assembly code including the introduced signal related opcodes.

The new high-level programming language features introduced by SigSolid are the following:

- **Signal declaration:** The statement *signal S(parmType)* declares a signal event *S* where *parmType* is the list of data types associated with the event parameters. SigSolid compiler translates the above statement into SigVM *createsignal* opcode.
- **Handler declaration:** using the keyword *handler* a function is declared as a signal handler. For instance, *function foo(...) handler { ... }* corresponds to defining a handler function *foo*.
- **Bind signal:** The statement *foo.bind(addr, S, gr, blk, sigR, sigM)* binds the handler function *foo* to a signal event *S* of a contract stored in the address *addr*. *gr* is the gas ratio. *blk*, *sigR*, and *sigM* are the parameters for the locking mechanism to enforce. The above statement translates into SigVM *bind* opcode.
- **Detach signal:** The statement *foo.detach(addr, S)* detaches the handler function *foo* from the signal event *S* of the contract stored in the address *addr*. The above statement translates into SigVM *detach* opcode.
- **Emit signal:** The statement *S.emit(parm).target(t).delay(d)* emits a declared signal event *S* where *parm* are the passed values of signal parameters, *t* is an array of addresses of the recipients intended by the emitted signal, *d* is a non-negative integer that specifies the number of blocks as the delay. The above statement translates into SigVM *emit* opcode.

SigVM is designed to prevent denial-of-service attacks in which an attacker triggers a large number of signal events to block regular transactions of a victim contract. Such attacks are possible only when the victim contract listen to an event from a contract deployed by the attacker in a way that blocks regular transactions. The victim contract can simply detach its handler function to stop listing to events from the attacker. Also, the contract locking mechanism in SigVM allows the victim contract to remove the locking partially by specifying a list of trusted addresses for which regular transactions are not blocked (using *sigR*, and *sigM* in signal binding).

5.2 SigChain

Our SigChain extends OpenEthereum [Technologies 2019] to support signal transaction execution. In particular, it extends the OpenEthereum EVM with the implementations of SigVM signal instructions. The execution of a signal transaction in SigChain is similar to that of a regular transaction. The main difference is that it starts at a handler function attached to emitted signal rather than a user specified function. The blockchain state is augmented with signal events related fields that we presented in Section 4. Also, it maintains a list of all accounts with pending signal transactions which provides an efficient method for miners to retrieve pending signal transactions.

In SigChain, a smart contract who registers a handler function will pay the transaction fee cost of the associated signal transaction. If the smart contract is unable to pay the fee of its signal transaction, the signal transaction execution will stay in the pending transaction pool until the contract obtains sufficient native tokens to cover the fee. Note that simple native token transfer transactions that interact with the fallback method of the contract are not affected by the contract locking mechanism by default.

Enforcing contract locking: In SigChain we extend the transactions pool implementation in OpenEthereum. In particular, when processing a block SigChain checks that for each: (1) regular transaction whether the transaction is allowed under the locking mechanism or the global pending signal transactions queue is empty; and (2) signal transaction that the transaction is indeed the next transaction in the queue. If these verification steps fail, the transaction will not be executed and will have no effect. Miners who pack the transaction will receive no transaction fee reward from the transaction. SigChain also enforces contract event locking at the boundary of inter-contract calls. It checks whether the caller and the called function are permitted under the locking mechanism or the pending signal s queue of the callee is empty. If the check fails, the whole transaction will be reverted with no effect and the miner will receive no transaction fee reward. For the delegate call opcode, the check is skipped because delegate calls do not change the state of the callee.

Remark 5.1. *Similar to locks in other programming languages, the locking mechanism in SigVM if not used properly can result in a deadlock. For instance, if two contracts exchange signal transactions and the handler function of each contract call a function in the other contract (through regular transaction) then this will result in a deadlock when both contracts block regular transactions while signal transactions did not finish. This is because each signal transaction will be awaiting for the corresponding regular transaction that it invoked which cannot proceed since the other contract is locked while processing its signal transaction. To avoid such issues smart contract developers are provided with the locking whitelist mechanism using the parameters* sigR *and* sigM *when binding signals. The whitelist will allow to execute specific functions from designated addresses to avoid deadlock.*

5.3 Gas Mechanism

To incentivize miners to pack signal transactions as soon as possible, signal transactions pay a higher gas price than regular transactions. In SigChain, we set the gas price of signal transactions to be *the average gas price of regular transactions in the block multiplied by one plus the gas incentive ratio set during handler binding*. This mechanism ensures that the gas price of signal transactions will always be competitive comparing to regular transactions. Miners will always be incentivized to pack signal transactions even in the presence of regular transactions with high fees. Because these regular transactions will raise the gas price of signal transactions, thus miners earn the profits by packing signal transactions instead of other regular transactions with low gas fees.

To avoid situations where the average gas price of regular transactions cannot be determined due to an overabundance of signal transactions, SigChain sets a separate gas limit for signal transactions to reserve room for regular transactions. SigChain limits the total gas consumption of signal transactions in a block to be less than $\frac{1}{10}$ of the block gas limit. We set this limit based on the current Ethereum ratio between contract internal and regular transactions of 1 : 9 [Etherscan 2021].

SigVM gas mechanism is designed to follow EIP-1559 protocol of Ethereum [Buterin et al. 2019] (implemented in London hard fork upgrade in August 2021). The EIP-1559 protocol introduces a burning mechanism of base fees in transactions to speed up the mining of native token Ether, incentivize it, and boost Ether's price by reducing the supply of Ether. In particular, in EIP-1559 protocol a base gas fee is fixed for transactions to be included in a block. Then, when transactions are executed the base fees are burned and removed from circulation. Using this protocol, SigVM gas mechanism is designed to prevent malicious miners from mining gas fees through signal transctions. Malicious miners cannot profit by raising the average gas price in a block, by generating (fake) transactions with high gas fees, resulting in a higher gas price for signal transactions. In particular, under EIP-1559 since the base fees of transactions are burned and not paid to miners, then the attack by malicious miners to raise the gas price of signal transactions will not be profitable.

6 EVALUATION

In this section we outline an empirical study of SigVM using 23 contracts from 13 decentralized applications (DApps) on Ethereum using SigSolid language described in Section 5.1. Our empirical study is driven by the following questions:

(1) Can SigVM enable truly decentralized contracts? Specifically, can we implement popular smart contracts from Ethereum in SigVM to eliminate off-chain centralized relay servers?
(2) Is SigVM easy to use? What is the development effort of implementing smart contracts in SigVM?
(3) What is the performance overhead of SigVM compared with Ethereum? What is the run time of transactions in SigVM?
(4) What is the gas consumption for transactions in SigChain?
(5) What kinds of security risks are associated with off-chain relay server solutions? Can the contract event lock mechanism in SigVM eliminate these risks?

We run the experiments on a 8 cores machine (2.6 GHz Intel Core i7) with 16 GB memory.

6.1 Benchmark Evaluation

We collect some benchmarks whose contracts need relay server to trigger and poke the corresponding transactions on Ethereum. Our benchmark set contains 23 contracts from 13 DApps deployed on Ethereum, including MakerDAO [Foundation 2019], Compound [Leshner and Hayes 2019], and Augur [Peterson et al. 2019]. Compound is a decentralized lending service platform where users could deposit one type of digital asset as collateral to borrow another type of digital asset from its shared pool. Augur is a decentralized prediction market where users bet on future results of real-world events. The supplementary material contains descriptions for the other DApps. In total, the 13 DApps manage more than $3B of digital assets on the Ethereum network, making up an essential part of the Ethereum economic ecosystem. We rewrite the contracts into SigSolid in order to eliminate relay server and evaluate its performance.

Table 1 presents the experimental statistics of our benchmark. Each DApp contains one or more contracts that are interdependent with each other but are potentially maintained by different groups of users. In our evaluation, we focus on 23 contracts (column *Contract*) that are relying on off-chain relay servers to invoke poke functions.

We reimplemented the benchmark contracts in SigSolid with the goal of eliminating off-chain relay servers from the picture. Because SigSolid is compatible with Solidity, our modification

		Solidity		**SigSolid**			
Application	**Contract**	*pf*	*loc*	*se*	*hf*	*loc'*	*diff*
MakerDAO	Median	1	175	1	0	177	2
	OSM	2	178	1	1	174	6
	Spot&Vat	1	116	0	1	116	5
Compound	Timelock	3	113	1	1	115	12
Augur	Universe	1	711	1	1	727	15
MultiSigWallet	MultiSig	1	256	1	1	270	19
Historical Price Feed	HistoricalPrice	1	564	1	1	555	4
Idex	Exchange	1	1223	1	1	1231	8
Aave	LendingPool	1	355	1	1	359	4
Metronome	Metronome	1	125	1	1	133	8
DAIHardFactory	DAIHard	1	505	1	1	533	29
KyberDxMarket	PriceFeed	1	201	1	0	203	2
	DSValue	2	22	1	1	26	4
	Medianizer	1	195	0	1	197	2
DutchExchange	PriceFeed	1	201	1	0	203	2
	DSValue	2	22	1	1	26	4
	Median	1	2604	0	1	2606	2
PriceFeed	PriceFeed	1	201	1	0	203	2
	DSValue	2	22	1	1	26	4
	Median	1	199	0	1	201	2
OracleInterface	PriceFeed	1	201	1	0	203	2
	DSValue	2	22	1	1	26	4
	Medianizer	1	483	0	1	485	2

Table 1. Empirical results. Characteristics of applications: lines of code (*loc*) and number of poke functions (*pf*). Characteristics of applications using signal events: number of signal events (*se*), number of handler functions (*hf*), lines of code (*loc'*), and lines of code difference (*diff*).

is merely replacing the poke functions with SigSolid signal event statements. We are able to reimplement all 23 contracts in SigSolid. Columns *se* and *hf* in Table 1 present the number of declared signal events and the number of declared handler functions in the new contract implementation, respectively. Column *diff* presents the number of lines difference between the new and old code.

By inspecting the line of codes and the line differences of all of contracts at Column *loc*, *loc'* and *diff*, we note that the difference is around 10 lines. The new code is typically simpler than the original code while the reimplementation effort is lightweight. We modified on average 3.17% of the contract code. Therefore, the work of modifications into SigSolid is feasible for smart contract developers.

6.2 SigChain Performance Evaluation

Now we present a performance evaluation of SigChain against the Ethereum blockchain. In this experiment, we selected four types of transactions: native currency transfer, blockchain state read transaction, regular poke transaction, and signal transaction. For blockchain state read transaction, we use the `Peek` function of the `Median` contract that reads the current price of digital asset (line 3 in Listing 1). For regular poke transactions, we invoke the `OSM` `Poke` function to update the price from `Median` (line 25 in Listing 1). For signal transaction, we use the signal event that invokes the

Blockchain	Normal transfer	Contract call	Poke transaction	Signal transaction
SigVM	$43.423\mu s$	$225.878\mu s$	$409.918\mu s$	$400.579\mu s$
Ethereum	$42.435\mu s$	$223.105\mu s$	$408.433\mu s$	-

Table 2. Average time for transactions execution on SigVM and Ethereum. $1\mu s = 10^{-6}s$.

handler function prUpdt (line 26 in Listing 1) that replaces the `OSM` `Poke` function. In the experiment, we execute each transaction type 1000 times on each blockchain.

In Table 2, we report the average execution time for four types of transactions. Note that the signal transaction in the fifth column is only available in SigChain.

For the three common transaction types supported by the two blockchains, i.e., normal transfer, contract call and poke transaction, the performance of SigChain closely matches Ethereum. The biggest overhead is $2.73\mu s$ for the blockchain state read transaction. Thus, we believe that SigChain does not cause a notable performance overhead over Ethereum. Indeed, the throughputs in terms of the number of transactions per second (TPS) that can be generated by both Ethereum test network (disabling the proof-of-work consensus) and SigChain reach the same rate of 98 TPS. Note that this TPS is considered even high for Ethereum main network which processes only 7 to 30 transactions per second because of the consensus protocol and block size restrictions.

Interestingly, the experiment shows that the signal transaction for price update is a bit faster under SigChain than the corresponding poke transaction under SigChain as well. This is because of the code simplifications introduced by using signal events (e.g., removing unnecessary guards statements, lines 27 and 31 in Listing 1). This shows that the SigSolid version of smart contract by removeing some unnecessary requirements, allows to decrease the run time and still guarantee the correctness and the security of the smart contract.

6.3 SigChain Gas Consumption Evaluation

We now present an evaluation of the gas mechanism for signal transactions in SigChain. In this experiment, we collect the gas informations for the 10 most recent poke transactions on Ethereum for four DApps. For each poke transaction type, we collect the average gas price over the corresponding blocks range and the average gas price over the 10 transactions. Then, we compute the average gas fees for each poke transaction by multiplying the total gas units consumed by the poke function and the average gas price over the 10 poke transactions. We also compute the the gas fees for each signal transaction by multiplying the total gas units consumed by the handler function and the average gas price over blocks range multiplied by $1 + 0.1$[2].

In Table 3, the column *Blocks* presents the blocks range containing the 10 transactions. Column *BAvGPr* presents the average gas price over the blocks range. Column *PTxAvGPr* presents the average gas price over the 10 transactions. Column *PTxG* presents the total gas units consumed by the poke function. Column *PTxAvGFe* presents the average gas fee over the 10 transactions, i.e., $PTxAvGFe = PTxG \times PTxAvGPr$. Column *STxG* presents the total gas units consumed by the handler function for signal transaction. The last column *STxGFe* presents the average gas fees for each signal transaction, i.e., $STxGFe = STxG \times BAvGPr \times 1.1$.

The results in Table 3 show that the increase in the gas fees for signal transactions over poke transactions is reasonable. This is because that most of signal transactions consume less gas than their counterparts (columns *STxG* and *PTxG* in Table 3). In particular, signal transactions allow to remove some guards statements (e.g., "require") that become useless. For instance, in the case of Compound some conditional checks in `executeTransaction` are removed because they are

[2]We assume that the gas rate is gr = 0.1.

Application	Function	PTxG	STxG	Blocks	BAvGPr	PTxAvGPr	PTxAvGFe	STxGFe
MakerDAO	Spot Poke	25571	26672	14444015-14444123	58.64	57.53	1471099.63	1720450.68
	OSM Poke	55881	54073	14442977-14444015	68.44	69.34	3874788.54	4070831.73
Compound	executeTr-ansaction	46090	45992	14402761-14430843	31.5	29.8	1373482	1593622.8
MultiSig-Wallet	executeTr-ansaction	49853	48913	12278821-13773042	205.41	226.35	11284226.55	11051941.26
Historical Price Feed	Poke	55963	54512	12117196-12175764	169.9	156.97	8784512.11	10187747.68

Table 3. Poking and Signal Transactions Gas Consumption. Characteristics of transactions: the name of the poke function invoked by the transaction (column *Function*), the total gas units consumed by calling the poke function (column *PTxG*), the total gas units consumed by the handler function (column *STxG*), the blocks ranges from which the 10 poke transactions were obtained (column *Blocks*), the average gas price over the blocks in the column *Blocks* (column *BAvGPr*), the average gas price over the 10 poke transactions (column *PTxAvGPr*), the average gas fees over the 10 poke transactions (column *PTxAvGFe*), and the gas fees for a signal transaction (column *STxGFe*). The unit for the gas price and fees columns is *GWei*.

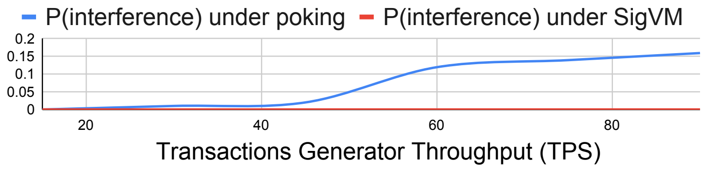

Fig. 5. Transaction interference rate.

guaranteed by SigChain. This allows to simplify the implementation and reduce gas consumption. Interestingly, the signal transaction for the MultiSigWallet application has less gas fees than its counterpart poke transaction which was surprising.

6.4 Eliminate Risks of Off-chain Relay Server

We next study the risk of regular transactions interfering with poke transactions when using off-chain relay servers. We also evaluate whether SigVM eliminates this risk.

Interfering Transactions: Using off-chain relay servers to simulate event-driven execution models may make the contract vulnerable to interfering transactions, i.e., a user may interact with a contract before a critical poke transaction. This may cause the contract to execute interference transactions in an incorrect state (e.g., liquidating assets with an incorrect asset price). Such interfering transactions can naturally occur during network congestion if the gas price of poke transactions are not high enough. They can also be caused by malicious miners who ignore transaction fees and prioritize such transactions intentionally.

We setup an experiment with the MakerDAO contracts to quantitatively illustrate this problem. In this experiment, we start a single-node Ethereum test network that generates blocks (disabling the proof-of-work consensus) with one block generated each 10 seconds. We developed a random *transactions generator* to generate three types of transactions, 1) simple transfer transactions that do not interact with MakerDAO, 2) MakerDAO transactions, and 3) poke transactions that attempt to update price for MakerDAO. In the experiment, the ratio of the three kinds of transactions is 10 : 4 : 1, respectively. The gas price for the three transactions follows the normal distribution with an average: 15, a standard deviation: 5, and a maximum: 50. Note that to facilitate experimentation, we changed the delay period for the price feed event `DPr` in MakerDAO from one hour to 10 seconds (i.e., one block delay).

We run the experiment multiple times and gradually increase the throughput of the *transactions generator* to 90 TPS to simulate the different congestion levels of a blockchain network. Each run lasts 5 minutes. Fig. 5 summarizes our results, showing the chance of a poke transaction being interfered by other MakerDAO transactions that are issued later than the poke transaction grows as we increase the *transactions generator* throughput. Interfering transactions occur starting from a transaction generation rate $\geq$ 20 TPS and goes as high as 0.15 when the *transactions generator* sends 80 TPS. We repeat the experiment with the new MakerDAO contracts in our SigChain platform by replacing poke transactions with the automatic signal event mechanism. The transaction interference never occurs because of our novel contract event lock mechanism.

Indeed, the results proves that the lock mechanism is effective for guarding the interests of contract participants. Attackers will send transactions to interfere with the online market before a price or a proposal is settled. Then the normal poking transactions become unreliable for normal users. Thus, the lock mechanism prevents such undesired and malicious contract interactions before the signal transaction is executed. It stabilizes the running process of online market.

6.5 Case Study Validation & Analysis:

We select the three applications MakerDAO, Compound, and Augur as case studies and we report our experience in analyzing the code of the new contracts. We validate that all three applications function correctly with the new contract code and they no longer depend on off-chain relay servers.

6.5.1 Case Study: MakerDAO. As described in Section 2, we reimplemented the three contracts in the oracle components of MakerDAO, eliminating the dependency on off-chain relay servers. The new contracts powered by SigVM ensure that the core MakerDAO engine will always operate with the updated price of its underlying digital asset. In addition, before the MakerDAO contract processes these price feed signal transactions, by lock mechanism, other transactions like liquidation and bidding will be put on-hold to avoid them operating with stale price information.

6.5.2 Case Study: Compound. Compound protocol is a decentralized market to lend or borrow assets. Users communicate with the Compound contracts to supply, withdraw, borrow and repay assets. In the protocol, the key parameters of the market, such as interest rate, risk model and underlying price, are managed by a decentralized government body in Compound [Leshner and Hayes 2019]. Any proposals of updating the parameters must be voted for approval and queued in `Timelock` contract. To give market participants time to react for any parameter changes, the queued proposals will be executed in the required delay specified in the proposals.

In the original Compound design, contract's Governor calls `queueTransaction` (line 13) to queue proposals. But to implement the desired delay, Governor or an off-chain relay server needs to send another poke transaction to call `executeTransaction` (line 27) right after the completion the delay period and before the proposal expiration time. `executeTransaction` first checks whether the proposal is queued and if the current timestamp falls in the required interval (lines 32-33). If so, the proposal is executed. This design is not desirable because the Governor or the off-chain relay server may fail and/or the poke transaction may not be packed in time.

Therefore, we reimplemented the `Timelock` contract in SigSolid and validated it with our SigChain platform. Listing 3 presents the simplified code snippet of `Timelock`. Lines preceded with "-" and "+" signs are changes we made to the code. Through such modification in SigSolid, we eliminate the dependency on poke transactions with the signal event mechanism. We define a new signal

```
1  contract Timelock {
2    struct LockedTx { //Information of a transaction
3      address target;uint txvalue;string signature;
4      bytes datain;uint user_delay;bool ready; }
5    mapping (bytes=>LockedTx) queuedTx;
6    //max delay when poking executeTransaction
7  - uint public constant GRACE_PERIOD = 14 days;
8  + uint public delay;// Base delay to execute
9  + address[] SigR; //allowed accounts
10 + bytes[] SigM; //accessible methods
11 + signal TimeUp(bytes); //signal event
12 + address[] SigT; //target handler addresses
13   function queueTransaction(address target,uint txval,
14     string memory signature,bytes memory data,uint userDelay)
15      public returns (bytes) {
16     // Hash transaction parameters as index
17     bytes txHash=keccak256(abi.encode(target,txval,signature,data,userDelay));
18     //Replace timestamp with offset delay
19 -   require(eta>=getBlockTimestamp().add(delay));
20 +   require(userDelay >= delay);
21     // Push the new transaction to queuedTx
22     queuedTx[txHash]=LockedTx(target,txval,signature,data,userDelay,true);
23     // Emit signal event with txHash to itself
24 +   SigT=[this.address];
25 +   TimesUp.emit(txHash).target(SigT).delay(userDelay);
26   // handler for executing tx
27 -  function executeTransaction(bytes txHash) public {
28 +  function executeTransaction(bytes txHash) handler {
29      //remove check that sender is admin
30 -    require(msg.sender == admin);
31      //remove checks that tx is in the correct period
32 -    require(getBlockTimestamp() >= eta);
33 -    require(getBlockTimestamp() <= eta + GRACE_PERIOD);
34       // use queuedTx[txHash] to do function call
35       ... }
36   constructor(address ad,uint del,address[] accs) public{
37 +  SigR = accs; //signal initializations
38 +  SigM = [abi.encode("queueTransaction(address,uint,
39 +           string,bytes,uint)"), ... ];
40    //bind handler to signal
41 +  executeTransaction.bind(this.address,TimeUp,0.1,true,SigR,SigM);
42   } ... }
```

Listing 3. Code snippets of Timelock.sol in Compound

event called TimeUp at line 11. The queueTransaction function will emit this event with the specified delay (line 25). Instead of being a public function, the executeTransaction function is declared as an event handler (line 28). When the contract is constructed, we bind executeTransaction as the handler for TimeUp events in the same contract. Thus, executeTransaction will be automatically executed once the specified delay time (measured in the number of blocks) has past after each TimeUp event. More significantly, many conditional checks in Solidity such as owner check, timestamp check are completely removed in SigSolid because signal transaction is automatically executed along with both the owner address and one target delay as discussed in Section 4. The requirements are totally guaranteed by the execution engine of SigVM. The removal of conditional checks decreases the total run time of the transactions and improves the security of the execution even though smart contract developers ignore the necessary checks.

```
1  contract Universe is IUniverse {
2      //daily block rate of the network
3      uint public dBlkR;
4  +   signal DSig(); //daily signal event
5  +   address[] SigT; //target handler addresses
6  +   address[] SigR; //allowed accounts
7  +   bytes[] SigM; //accessible methods
8      //handler function
9  +   function Update() handler {
10 +    sweepInterest();
11      //signal emission
12 +    SigTargets = [this.address];
13 +    DSig.emit(this.address).target(SigT).delay(dBlkR); }
14     function sweepInterest() public returns (bool) {...}
15 +   function setBlkRate(uint _dBlkR) public {
16 +    require(_dBlkR >= MIN_RATE);
17 +    dBlkR = _dBlkR; }
18 +   function start_emit() public {
19 +    SigT = [this.address]; //emit a signal immediately
20 +    DSig.emit().target(SigT).delay(0); }
21     constructor(Augur _augur, ..., uint _dBlkR) public {
22 +    dBlkR = _dBlkR; //set handler execution period
23      //update SigRoles and SigMethods
24 +    ...
25      //bind handler to signal
26 +    Update.bind(this.address,DSig,0.1,true,SigR,SigM);
27       ... } ... }
```

Listing 4. Code snippets of Universe.sol in Augur

We deployed the modified `Timelock` contract together with the rest of Compound contracts in SigChain. The deployed Compound application in SigChain operate successfully and there is no need to run off-chain server to poke the `executeTransaction` function anymore.

6.5.3 *Case Study: Augur.* Augur is a decentralized prediction market where users can bet on future real-world outcomes [Peterson et al. 2019]. It follows 4 stages: creation of the market, trading, reporting outcomes and prediction settlement. During reporting stage, a reporter reports results of a real-world event by staking REP tokens on one of the outcomes of the market. If a Dispute Bond is reached on an alternative outcome, the Tentative Winning Outcome changes to the new alternative outcome. This process happens every 24 hours.

The contract that implements this process in Augur is `Universe`. Every 24 hours, the `sweepInterest` function in `Universe` has to be executed to finalize the last round results and to start a new round if required. Because this functionality is not feasible to implement in EVM, Augur instead relies on external users to periodically poke `sweepInterest`.

Listing 4 presents the code snippet of our `Universe` contract implementation in SigSolid. We define a delay signal event called `DSig` at line 5 and we define a handler function called `Update` for this event (lines 10-14). The handler function calls `sweepInterest`. It also recursively emits the `DSig` event with a delay of one day (in the number of blocks) at the end (line 14). Therefore once the first `DSig` event is triggered (via the `start_emit` function), the `Update` function will be automatically invoked once every 24 hours.

We deployed the new `Universe` contract together with the rest of contracts in Augur in our SigChain platform. The deployed version of Augur operates successfully and the `sweepInterest` function is automatically called every 24 hours as expected. This example highlights the expressiveness of SigSolid powered by SigVM. This recursive event handler pattern enables developers to create customized infinite timer ticks conveniently. The desired handler function will be automatically and periodically invoked as signal transactions by the execution engine of SigVM, as long as the contract has enough native tokens to pay for the transaction fee of the recurring transactions.

7 RELATED WORK

Event-driven proxy services: On Ethereum, there are a number of proposals involving proxy service of function executions in respondence to event emission. EventWarden [Li and Palanisamy 2020] is a proxy service for a user to create a smart contract specifying events to listen, the function handler to call when an event occurs, and the service fee for the executor. Executors of the system monitor the event log, invoke the specified function, and earn a service fee in return. Similarly, Ethbase [Ethereum 2018] provides a registry smart contract where users can deposit and subscribe to an automatic function invocation service. When the specified event is seen in the log, miners invoke the callback function accompanied by a proof of the event emission to the registry. Both EventWarden and Ethbase are variants of off-chain relay server solutions, although they provide incentives on chain via smart contracts so that any external user could act as an off-chain relay server to send poke transactions to claim the reward. They share similar weakness as the off-chain relay server solution, i.e., the reliance on external users to send poke transactions timely and their willingness to pay high enough fees for these poke transactions to be packed timely. Different from these approaches, our solution modifies the blockchain system to enable native support of an event-driven execution. SigVM brings the whole execution process on-chain and completely eliminates the dependency on off-chain relay actors.
Smart contracts programming: In addition to Solidity, a number of new languages have been designed for smart contracts programming, including, Vyper [Team 2020], Simplicity [O'Connor 2017], Liquidity [OCamlPRO 2020], Sophia [æernity dev team 2020], Move [Blackshear et al. 2020], DeepSEA [Sjöberg et al. 2019], and Obsidian [Coblenz et al. 2019]. Unlike SigVM, the above languages do not support an event-driven execution model. Interestingly, because the event-driven execution model is highly demanded and many smart contracts are unsecurely simulating such an execution model via off-chain relay servers, SigVM also improves the security of these contracts by eliminating the need of such insecure practice. Also the aforementioned languages either introduces new syntaxes that leads to new learning curve for developers or suppress expressiveness in order to gain security guarantees. On the other hand, SigSolid is a practical extension on Solidity with minor changes in the syntax while improving security and usability of the language.

Recent smart contract languages such as RhoLang [Cooperative 2019], Scilla [Sergey et al. 2019], and Nomos [Das et al. 2021] replace inter-contract function calls with message-passing mechanisms to eliminate bugs like re-entrance. Although those message-passing mechanisms are also asynchronous, they behave otherwise like function calls and cannot implement the desired event-listener model to eliminate off-chain relay servers. In fact, it would be difficult to implement many DeFi contracts like MakerDAO in those languages, because these contracts depend on synchronous inter-contract function calls. Making a function call asynchronous via message-passing may make such a contract vulnerable to malicious manipulation attacks that front-run the asynchronous massage-passing transaction.

General purposes programming languages such as JavaScript support event-driven asynchronous programming. However, they are not commonly used to write smart contracts.
Security and correctness validation: Another path taken by many to improve the security of smart contract programming is through static verification and runtime validation tools. There is a rich body of literature on detecting vulnerabilities in smart contracts through static analysis and modular verification [Bhargavan et al. 2016; Hildenbrandt et al. 2018; Kalra et al. 2018; Luu et al. 2016; Nikolić et al. 2018; Permenev et al. 2020]. For instance, ith , Oyente [Luu et al. 2016] uses

symbolic executions to verify smart contracts against various attacks including re-entrance and mishandled exception attacks. Verx [Permenev et al. 2020] uses delayed abstraction to detect and verify temporal safety properties automatically. Solythesis [Li et al. 2020a] inserts runtime checks to enforce customized validation rules. KEVM [Hildenbrandt et al. 2018] defines a formal semantics of EVM in $\mathbb{K}$ and verifies smart contracts against user-defined specifications. These techniques enhance security of smart contract programming mostly by preventing unintended behaviors in smart contract codes and cannot be used to eliminate the dependency on off-chain relay servers.
Scheduled transaction execution: Bitcoin supports a native mechanism called timelocks [Antonopoulos 2017] to delay transaction execution. The transaction-level timelock feature can be utilized by specifying an execution delay with the nLocktime field. Additional timelock features, Check Lock Time Verify (CLTV) and Check Sequence Verify (CSV), were introduced later to the scripting language. CLTV limits the availability of the associated Unspent Transaction Output (UTXO) until a certain age. CSV utilizes the the value of nSequence, a transaction field, to prevent mining of a transaction until the time limit specified for the UTXO is met. These features are proven to be useful in layer 2 designs, such as state channels and the Lightning Network. The timelock features provides a convenient way to delay transaction execution on Bitcoin. However, with the lack of programmability on Bitcoin and the difficulty of generalizing such a design to non-UTXO networks, the usability of such a design is limited. Another implementation of delayed or periodic transaction is to setup relay servers through the network client as we discussed before this is not desirable.

8 CONCLUSION

As smart contracts become more complicated and inter-dependent with each other, the event-driven execution model is an increasingly demanded feature. Unsatisfied by current blockchain virtual machines, smart contracts start to use unreliable mechanisms such as off-chain relay servers to insecurely simulate event-driven execution. Unlike other software applications smart contracts also pose new challenges for integrating an event-driven execution model, e.g., decentralized runtime environment and incentive mechanisms. SigVM provides the first blockchain virtual machine with an integrated solution to natively enable event-driven execution models on-chain. Such automatic transaction execution eliminates the poking of relay servers. It reduces network congestion. Also, the contract locking mechanism avoids unexpected contract interactions before signal transaction is executed. For instance, for the MakerDAO market regular transactions will be disabled before the spot prices are updated. In addition, under the new gas price mechanism for signal transactions, the extra gas price ratio will incentivize miners to execute signal transactions ahead of regular transactions. It will allow to prioritize signal transactions and execute them in the desired block delay.

SigVM is developed as an extension of EVM allowing an easy integration into blockchains that work with EVM-based execution environment, e.g., Avalanche, Conflux, and Near, to maintain compatibility (due to the popularity of the EVM stack for developers).

REFERENCES

æernity dev team. 2020. The Sophia Language. https://github.com/aeternity/aesophia/blob/lima/docs/sophia.mdl

Andreas M. Antonopoulos. 2017. *Mastering Bitcoin: Programming the Open Blockchain* (2nd ed.). O'Reilly Media, Inc.

Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. 2016. Formal Verification of Smart Contracts: Short Paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. 91–96. https://doi.org/10.1145/2993600.2993611

Sam Blackshear, Evan Cheng, David L. Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Rain, Dario Russi, Stephane Sezer, Tim Zakian, and Runtian Zhou. 2020. Move: A Language With Programmable Resources. https://developers.libra.org/docs/assets/papers/libra-move-a-language-with-programmable-resources/2020-05-26.pdf

Vitalik Buterin. 2014. Ethereum: A next-generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper.

Vitalik Buterin, Eric Conner, Rick Dudley, Matthew Slipper, Ian Norden, and Abdelhamid Bakhta. 2019. Fee Market Change for ETH 1.0 Chain. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1559.md

ChainLink. 2020. ChainLink: Using Price Feeds. https://docs.chain.link/docs/get-the-latest-price

Michael Coblenz, Reed Oei, Tyler Etzel, Paulette Koronkevich, Miles Baker, Yannick Bloem, Brad A. Myers, Joshua Sunshine, and Jonathan Aldrich. 2019. Obsidian: Typestate and Assets for Safer Blockchain Programming. arXiv:1909.03523 [cs.PL]

RChain Cooperative. 2019. Rholang. https://github.com/rchain/rchain/tree/master/rholang

A. Das, S. Balzer, J. Hoffmann, F. Pfenning, and I. Santurkar. 2021. Resource-Aware Session Types for Digital Contracts. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, Los Alamitos, CA, USA, 111–126. https://doi.org/10.1109/CSF51468.2021.00004

Corentin Denoeud. 2020. What Is A Transaction Relayer And How Does It Work? https://hackernoon.com/what-is-a-transaction-relayer-and-how-does-it-work-bd1q3ywa

Ethereum. 2020. The Solidity Contract-Oriented Programming Language. https://github.com/ethereum/solidity

Planet Ethereum. 2018. Ethbase. https://github.com/planet-ethereum/ethbase

Etherscan. 2021. The Ethereum Blockchain Explorer. https://etherscan.io/

Vitalik Buterin Fabian Vogelsteller. 2015. "EIP-20: ERC-20 Token Standard," Ethereum Improvement Proposals. https://eips.ethereum.org/EIPS/eip-20

The Maker Foundation. 2019. The Maker Protocol: MakerDAO's Multi-Collateral Dai (MCD) System. https://makerdao.com/en/whitepaper/

The Maker Foundation. 2020. The Market Collapse of March 12-13, 2020: How It Impacted MakerDAO. https://blog.makerdao.com/the-market-collapse-of-march-12-2020-how-it-impacted-makerdao/

The Near Foundation. 2021. The NEAR White Paper. https://near.org/papers/the-official-near-white-paper/

E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu. 2018. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. 204–217. https://doi.org/10.1109/CSF.2018.00022

Sukrit Kalra, Seep Goe, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *Proceedings of the 25th Network and Distributed Systems Security (NDSS) Symposium 2018*. https://doi.org/10.14722/ndss.2018.23092

Robert Leshner and Geoffrey Hayes. 2019. Compound: The Money Market Protocol. https://compound.finance/documents/Compound.Whitepaper.pdf

Ao Li, Jemin Andrew Choi, and Fan Long. 2020a. Securing Smart Contract with Runtime Validation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 438–453. https://doi.org/10.1145/3385412.3385982

Chenxing Li, Peilun Li, Dong Zhou, Zhe Yang, Ming Wu, Guang Yang, Wei Xu, Fan Long, and Andrew Chi-Chih Yao. 2020b. *A Decentralized Blockchain with High Throughput and Fast Confirmation.* USENIX Association, USA.

Chao Li and Balaji Palanisamy. 2020. EventWarden: A Decentralized Event-driven Proxy Service for Outsourcing Arbitrary Transactions in Ethereum-like Blockchains. In *Proceedings of 27rd IEEE International Conference on Web Services*.

Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 254–269. https://doi.org/10.1145/2976749.2978309

Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 653–663. https://doi.org/10.1145/3274694.3274743

OCamlPRO. 2020. Liquidity Documentation. https://www.liquidity-lang.org/doc/

Russell O'Connor. 2017. Simplicity: A new language for blockchains. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*. 107–120. https://doi.org/10.1145/3139337.3139340

Michael Oved and Don Mosites. 2017. Swap: A Peer-to-Peer Protocol for Trading Ethereum Tokens. https://www.airswap.io/whitepaper.htm

A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev. 2020. VerX: Safety Verification of Smart Contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1661–1677. https://doi.org/10.1109/SP40000.2020.00024

Jack Peterson, Joseph Krug, Micah Zoltu, Austin K. Williams, and Stephanie Alexander. 2019. Augur: a Decentralized Oracle and Prediction Market Platform (v2.0). https://www.augur.net/whitepaper.pdf

provable.xyz. 2019. Provable: Ethereum Quick Start. https://docs.provable.xyz/#ethereum-quick-start

Kevin Sekniqi, Daniel Laine, Stephen Buttolph, and Emin Gün Sirer. 2020. Avalanche Platform. https://www.avalabs.org/whitepapers

Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. 2019. Safer smart contract programming with Scilla. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.

Vilhelm Sjöberg, Yuyang Sang, Shu-chun Weng, and Zhong Shao. 2019. DeepSEA: A Language for Certified System Software. (2019). https://doi.org/10.1145/3360562 Article No: 136.

Vyper Team. 2020. Vyper. https://vyper.readthedocs.io/en/stable/

Parity Technologies. 2019. OpenEthereum. https://www.parity.io/ethereum/