SigRec: Automatic Recovery of Function Signatures in Smart Contracts

Ting Chen, Zihao Li, Xiapu Luo, Xiaofeng Wang, Ting Wang, Zheyuan He, Kezhao Fang, Yufei Zhang, Hang Zhu, Hongwei Li, Yan Cheng, and Xiaosong Zhang

Abstract—Millions of smart contracts have been deployed onto Ethereum for providing various services, whose functions can be invoked. For this purpose, the caller needs to know the *function signature* of a callee, which includes its function id and parameter types. Such signatures are *critical* to many applications focusing on smart contracts, e.g., reverse engineering, fuzzing, attack detection, and profiling. Unfortunately, it is challenging to recover the function signatures from contract bytecode, since neither debug information nor type information is present in the bytecode.To address this issue, prior approaches rely on source code, or a collection of known signatures from incomplete databases or incomplete heuristic rules, which, however, are far from adequate and cannot cope with the rapid growth of new contracts. In this paper, we propose a novel solution that leverages how functions are handled by Ethereum virtual machine (EVM) to automatically recover function signatures. In particular, we exploit how smart contracts determine the functions to be invoked to locate and extract function ids, and propose a new approach named *type-aware* symbolic execution (TASE) that utilizes the semantics of EVM operations on parameters to identify the number and the types of parameters. Moreover, we develop SigRec, a new tool for recovering function signatures from contract bytecode without the need of source code and function signature databases. The extensive experimental results show that SigRec outperforms all existing tools, achieving an unprecedented 98.7% accuracy within 0.074 seconds. We further demonstrate that the recovered function signatures are useful in attack detection, fuzzing and reverse engineering of EVM bytecode.

Index Terms—smart contract, function signature, Ethereum, automatic recovery, type-aware symbolic execution.

1 Introduction

More than 30 million smart contracts have been deployed on Ethereum, the largest contract-hosting blockchain. An Ethereum *smart contract* is a program (i.e., a collection of code and data) running on Ethereum virtual machine (EVM), which can be accessed through its address in Ethereum [1]. A smart contract is typically written in a high-level language (e.g., Solidity [2]) and then compiled into EVM bytecode. After the bytecode is deployed, its public and external functions can be invoked. To invoke a contract function, the caller needs to know its function signature which consists of a function id and the list of parameter types [3][1]. A *function id* refers to the first 4 bytes in the Keccak-256 hash of a function name and the list of parameter types [3].

Function signatures play a critical role in many applications focusing on smart contracts because their functions (in terms of function signatures) need to be first identified before their behaviors can be evaluated. For example, some studies recognize wallet contracts [4], token contracts [5], [6], [7], and the contracts for Ethereum name service [6] based on function signatures. §6 shows that function signatures can be used to enhance the results of reverse engineering results contract bytecode by adding types, meaningful variable names and simplifying the code for accessing parameters. Besides, knowing the parameter list of a target function, a fuzzer can strategically mutate the test cases [8], [9] for better vulnerability detection. Our experimental results (§6) show that function signatures enable ContractFuzzer [8] to find 23% more bugs than it could without these signatures. Function signatures are also important to the detection of attacks against smart contracts. A prominent case is the *short address* attack that exploits an EVM vulnerability in handling a malformed actual argument of the address type, which is shorter than a valid address [10]. We develop ParChecker, a new tool based on the recovered function signatures to detect malformed actual arguments including short address attacks (§6).

Challenges in signature discovery. It is challenging to recover function signatures from bytecode since there is no debug information in contract bytecode and the bytecode is *untyped*, i.e., parameters are kept in 256-bit words without type information [11]. All existing approaches fail to effectively recover function signatures (§5.6). More precisely, an intuitive method is to extract the function signatures from the source code of smart contracts, which, however, is only available for a small proportion of contracts ($< 1\%$ until 2017 [12]). Another method retrieves signatures from existing databases, like Ethereum Function Signature Database (EFSD) [13], through function ids. For example, Gigahorse [14], Eveem [15], and Online Solidity Decompiler (OSD) [16] rely on EFSD, while EVM Bytecode Decompiler

- *Ting Chen (brokendragon@uestc.edu.cn), Zheyuan He, Kezhao Fang, Yufei Zhang, Hang Zhu, Hongwei Li and Xiaosong Zhang are with University of Electronic Science and Technology of China, China.*
- *Zihao Li and Xiapu Luo are with Hong Kong Polytechnic University, Hong Kong. (Corresponding Author: Xiapu Luo.)*
- *Xiaofeng Wang is with Indiana University, Bloomington, USA.*
- *Ting Wang is with Pennsylvania State University, USA.*
- *Yan Cheng is with Ant Group, China.*

1. A function signature defined by Ethereum includes a function name and the list of parameter types [3]. Since function id instead of function name is required to invoke a contract function, in this paper we just consider function id, parameter number and parameter types.

(EBD) [17] and JEB [18] maintain their own databases. However, these databases are incomplete, only covering 31.7% of the function signatures in the wild (Supplementary material A) and quickly become out of date without constantly updating.

Alternatively, one could reverse engineering a function id by enumerating all possible parameter type combinations to compute a hash, together with a possible function name. Given the huge space of possible combinations, this attempt is fragile in general. Abi Decompiler [19] adopts this approach, which only covers 12.3% of function signatures (Supplementary material A). Finally, one could derive function signatures using some heuristics: Gigahorse infers the number of parameters based upon the number of items that a function pops from stack [14]; Eveem regards a type as an array if its layout contains an *offset* field and a *num* field. Although this direction is promising, existing approaches turn out to be ineffective due to the incompleteness of their heuristics. For example, Eveem can only achieve an accuracy of 58.1% and 18.3% to infer the function signatures in closed-source smart contracts and synthesized contracts, respectively (§5.6), even it leverages an existing database, EFSD.

Our work. In this paper, we propose a new solution called `SigRec` to automatically recover function signatures from EVM bytecode compiled from two mainstream compilers (i.e., Solidity and Vyper) for smart contracts *without* the need of source code and other databases. Our key observation is that even in the absence of type information, the way EVM bytecode handles a function call and its inputs uniquely characterizes different parameter types. For example, a uint32 argument will be extracted using a 32-bits mask applied to the input data (R11, §3.4). Based upon this observation, we first generalize the semantics of such type-related operations into rules (§3). Then, we design *type-aware* symbolic execution (TASE) to explore the EVM instructions that manipulate parameters, and use the rules for inferring the types of parameters (§4). Since the EVM instructions handling parameters are typically near a function's entry point, TASE can handle them very effectively and efficiently to keep up with the ever-growing Ethereum smart contracts.

Studies on reverse engineering of variable types from binary executables by leveraging the semantics of instructions [20], [21], [22], [23], [24], [25], [26], [27], [28], [29] are related to our work. However, TASE has several differences with existing approaches. First, to recover complicated types that cannot be handled by existing techniques, TASE infers how variables are affected by parameters through symbolic execution (SE). For example, to identify a multidimensional array, TASE checks whether the read location is calculated by adding the value of the *offset* field to the base (§3.2). Second, semantic knowledge summarized from binary code cannot be applied to EVM bytecode, due to architecture differences (§8) and unique parameter types defined in smart contracts, e.g., 256-bit integers (no more than 64 bit in x64 binaries), the 20-byte address type, and reversed array notation (§2.3). New knowledge, therefore, needs to be discovered to support type inference through TASE. Third, by exploiting the fact that the code responsible for handling parameters is usually around a function's entry point, TASE runs extremely fast (0.074s to recover one function signature on average, §5.4).

We conduct extensive experiments to evaluate `SigRec` and compare it with all existing approaches. First, we collect all 119,404 unique (i.e., duplicates are eliminated) open-source smart contracts on Ethereum, which include 210,869 public/external functions with unique function signatures, and use them to evaluate the accuracy of `SigRec`. The experimental result shows that `SigRec` achieves an average accuracy of 98.7% (§5.2), and the accuracy never goes below 96% across all compilers (from V 0.1.1 to V 0.8.0) with or without optimization (§5.3). Second, we compare `SigRec` with all existing approaches, i.e., OSD [16], EBD [17], JEB [18], Gigahorse [14] and Eveem [15]. The experimental results show that `SigRec` correctly recovers much more signatures, outperforming them by at least 22.5 %, 40.1% and 80.5% in processing open-source, closed-source and synthesized smart contracts, respectively (§5.6). Manual investigation shows that for many function signatures, existing approaches report wrong types, produce *nonexistent* types, output an *unspecific* type, add *nonexistent* parameters, miss parameters or fail to generate function signatures (§5.6). Third, we apply `SigRec` to the bytecode of all 37,009,570 smart contracts deployed on Ethereum with 47,329,149 public/external functions, and found that it only takes 0.074 seconds on average to recover a function signature (§5.4).

We further demonstrate how the recovered function signatures can enable the detection of stealthy short address attacks, empower the state-of-the-art fuzzer to discover more vulnerabilities, and enhance the result of reverse engineering contract bytecode (§6).

Contributions. The major contributions of the paper are summarized as follows:

• We propose a novel solution that exploits the semantics of EVM instructions to correctly and efficiently recover function signatures from the bytecode of smart contracts.

• We develop `SigRec` based on our new solution, and conduct extensive experiments to evaluate it. The experimental results show that `SigRec` achieves nearly 100% accuracy in signature recovery, under different compiler versions, within 0.074 seconds on average, and it significantly outperforms existing methods. We have deployed `SigRec` as an online web service http://bit.ly/SigRecWS and will release its code after paper publication.

• We use three important applications to demonstrate the usage of recovering function signatures, including attack detection, fuzzing and reverse engineering of EVM bytecode.

2 Accessing Patterns of Parameters

This section first introduces how function invocation is performed in EVM and then elaborate more on the EVM instructions and variables involved in a function invocation. The concepts of account, bytecode of smart contracts, and public/external function can be found in Supplementary material B.

Function invocation. To invoke a public/external function, a message will be sent by an EOA account or a smart contract account [30]. The message contains the address of the smart contract whose function will be invoked and the *call data* field which indicates the function to be invoked and carries actual arguments. The first 4 bytes of the call

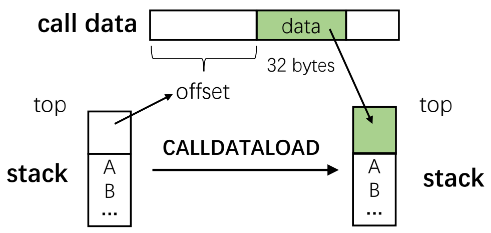

Fig. 1. The process of executing a CALLDATALOAD

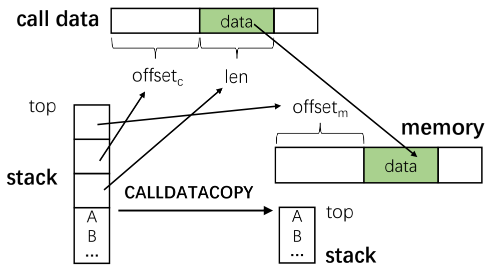

Fig. 2. The process of running a CALLDATACOPY

data is the function id of the function being called, which is followed by the arguments [3]. For example, to call a function whose signature is "transfer(address, uint256)", the call data begins with its function id 0xa9059cbb, followed by two arguments, including an address and a 256-bits unsigned integer.

2.1 EVM Instructions to Read the Call Data

CALLDATALOAD. CALLDATALOAD reads 32 bytes into the top of the stack [30]. As shown in Fig. 1, CALLDATALOAD first reads the top item from the stack, which is the offset used to locate the data. After that, the top item of the stack is removed. It then reads 32 bytes data from the call data starting from the offset. Finally, the data is pushed to the stack.

CALLDATACOPY. CALLDATACOPY reads variable-length data from the call data into memory [30]. As shown in Fig. 2, CALLDATACOPY consumes top three stack items. The second item denotes the offset ($offset_c$) in the call data from where the data will be copied. The first item is the offset ($offset_m$) in the memory to where data will be copied. The third item indicates the data length.

2.2 Accessing Function ID

To invoke a public/external function in a smart contract, the caller should provide the *function id*. The callee extracts it from the call data to determine the function to be invoked by executing a CALLDATALOAD with the offset being 0 to read 32 bytes from the beginning of the call data, whose highest 4 bytes is the function id. Then, the callee uses a DIV instruction (i.e., unsigned integer division [30]) or an SHR instruction (i.e., bitwise right shift [31]) to move the function id to the lowest 4 bytes.

2.3 Accessing Different Types of Parameters

Our work support two mainstream compilers, Solidity and Vyper which have different parameter types.

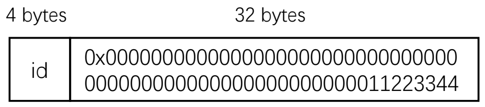

Fig. 3. The call data layout of a uint32, 0x11223344

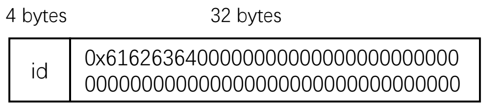

Fig. 4. The call data layout of a bytes4, 'abcd'

2.3.1 Solidity

We classify all parameter types supported by Solidity into five categories: basic types, array, bytes and string, and struct.
1. Basic Types: There are five basic types.
(1) uint$\langle M \rangle$, $8 \leq M \leq 256, M\%8 == 0$. uint$\langle M \rangle$ is an unsigned integer of M bits [3]. If a public/external function has a uint$\langle M \rangle$ type parameter, it will be extended on the higher-order (left) side with zeros to make the length be 32 bytes [3]. Fig. 3 shows the call data layout of a public/external function with one uint32 argument whose value is 0x11223344. The call data begins with a 4-byte function id, followed by the extended 32-bytes value. To read a uint$\langle M \rangle$, the smart contract executes a CALLDATALOAD with the offset indicating the start of the extended value. After that, the extended value will be masked by an AND instruction, and the result is the uint$\langle M \rangle$ argument before padding. The masking is not needed to access a uint256 because it is not extended.

(2) int$\langle M \rangle$, $8 \leq M \leq 256, M\%8 == 0$. int$\langle M \rangle$ is a signed integer of M bits [3]. The layout and accessing pattern of an int$\langle M \rangle$ is similar to that of a uint$\langle M \rangle$, except that a SIGNEXTEND rather than an AND is used to mask an extended value because SIGNEXTEND is responsible for sign extension [30]. Similarly, the SIGNEXTEND is not needed to access an int256 because the highest bit of an int256 indicates the sign. To distinguish an int256 from a uint256, both of which are not extended, we leverage some instructions (e.g., SDIV) that can only access signed integers.
(3) address. Every account has a unique 20-bytes address [30]. We find that the call data layout of an address and reading an address from the call data are the same as a uint160. To differentiate them, we exploit the fact that mathematic operations can involve a uint160 rather than an address.
(4) bool. It can be true or false. The layout and reading a bool from the call data are similar to that of a uint$\langle M \rangle$, except that two consecutive ISZEROs are used for masking. An ISZERO pushes a one to the stack if the top stack item is zero and a zero otherwise [30]. Hence, two consecutive ISZEROs push a one to the stack if the top stack item is not zero, otherwise push a zero.
(5) bytes$\langle M \rangle$, $0 < M \leq 32$. A bytes$\langle M \rangle$ is a byte sequence of M bytes [3]. If a public/external function has a bytes$\langle M \rangle$ type parameter, it will be extended on the lower-order (right) side with zero to make the length be 32 bytes [3]. Fig. 4 shows the call data layout of a public/external function with one bytes4 argument whose value is 'abcd'. A

CALLDATALOAD is needed to read a bytes$\langle M\rangle$ from the call data, and an AND is used for masking. After masking, the higher-order bytes of a bytes$\langle M\rangle$ is retained. By contrast, for a uint$\langle M\rangle$, the masking retains the lower-order bytes since a uint is extended on the higher-order side. Since bytes32 and uint256 are not extended, we differentiate them by exploiting the fact that a BYTE instruction is used for accessing a single byte of a bytes32, while an AND masks a uint256 for the same purpose.

2. Array: An array hosts array items of the same basic type. An array can be static, dynamic or nested. The size of a static array is known during compilation whereas the size of a dynamic array and a nested array is determined by the actual argument at runtime.

(1) Static array. We denote it as T$[X_1]...[X_n]$, where T is one of the basic types, X_1, ..., X_n are constant numbers known in compilation, and n is the dimension. If $n > 1$, T$[X_1]...[X_n]$ is a multidimensional array. The size of a static array is known in compilation since the number of items in each dimension is fixed. As a basic type, each array item is extended through the process described above. Different from other languages (e.g., C), the notation of an array in EVM is reversed and the access is in the opposite direction of the declaration [32]. For example, uint256[3][2] *x* means an array *x* of two arrays of three uint256, and *x*[1][0] accesses the first item of the second uint256[3]. All items of a static array are stored consecutively in the call data. Fig. 5 shows the call data layout of a public/external function with one uint256[3][2] argument.

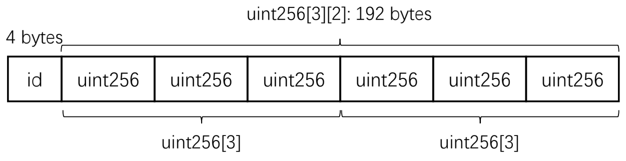

Fig. 5. The call data layout of a static array, uint256[3][2]

The accessing patterns of a static array are different for public and external functions.

(a) *Public function.* Its static array will be read into the memory by one or more CALLDATACOPYs. Then, each array item is accessed by an MLOAD, which reads 32 bytes from the memory to the stack [30]. We find that a nested loop is used to read parameters from the call data, and the level of the nested loop is equal to the dimension of the static array minus one. In the most inner layer, a CALLDATACOPY is executed. A CALLDATACOPY can read a one-dimensional static array.

Listing 1 shows the pseudocode (we do not show the EVM instructions for the ease of presentation) to read a uint256[3][2] from the call data to the memory. The execution of each CALLDATACOPY copies the array of the lowest dimension (Line 3). The loop guard (Line 2) determines how many times the CALLDATACOPY should be executed. The loop guard is compiled into an LT instruction, which checks whether the current loop count is smaller than the item number of the highest dimension.

```
1  i = 0;
2  while (i < 2){ //how many uint256[3] needs to copy
3    calldatacopy(uint256[3]);
4    i++;
5  }
```

Listing 1. Pseudocode for reading a uint256[3][2] parameter to memory

(b) *External function.* Its static array will not be entirely copied. Instead, array items will be read from the call data to the stack using CALLDATALOAD on demand. We find that if the smart contract is compiled without optimization or the index of the array is a variable, the CALLDATALOAD is preceded by bound checks to prevent array overrun. For instance, before accessing the item *x*[i][j] of the array *x*[3][2], two bound checks $i < 2$ and $j < 3$ are executed. Only after they are passed, *x*[i][j] can be accessed. Such runtime bound check is not needed if the array index is a constant and the smart contract is compiled with optimization due to the compile-time bound checks.

(2) Dynamic array. We denote it as T$[X_1]...[X_{n-1}][]$, where T is one of basic types, X_1, ..., X_{n-1} are constant numbers known in compilation, n is the dimension. The size of a dynamic array is unknown in compilation because the item number of the highest dimension is unknown. Instead, the item number should be provided in the call data so that the invoked function can know it at runtime. Fig. 6 shows the call data layout of a dynamic array uint256[3][], which is the first parameter, and the actual argument is uint256[3][2]. A 32-bytes value, i.e., the *offset* field, is located right after the function id, and this value is an offset relative to the first byte after the function id. The 32 bytes, i.e., the *num* field, pointed by the offset stores the number of items in the highest dimension. All array items are located after the *num* field.

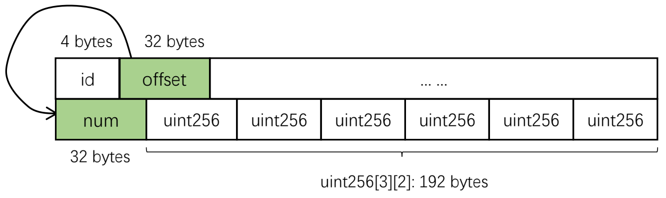

Fig. 6. The call data layout of a dynamic array uint256[3][], and the actual argument is uint256[3][2]

The accessing patterns of a dynamic array are different for public and external functions.

(a) *Public function.* Its dynamic array will be read into the memory by one or more CALLDATACOPYs. More precisely, the smart contract uses a CALLDATALOAD to read the offset (i.e., the *offset* field), and another CALLDATALOAD to read the item number of the highest dimension (i.e., the *num* field). Then, the item number will be read from the stack to the memory by an MSTORE, which saves 32 bytes from the stack into the memory [30]. After that, the smart contract uses a nested loop to copy all array items into the memory right after the item number. Only one CALLDATACOPY is needed to read a one-dimensional dynamic array.

(b) *External function.* Two CALLDATALOADs are used to read the offset and the item number of the highest dimension. Different from the process for public functions, an array item will be read from the call data to the stack by a CALLDATALOAD on demand. The operand of the CALLDATALOAD, i.e., the location from where to copy, is computed from the value of the *offset* field at runtime, and the result contains the multiplication of 32, because each array item is extended to 32 bytes. For example, the location of the third array item is computed by *offset*$+4+32+2\times32$ (the function id is of 4 bytes, an item number is of 32 bytes, the first two array items are of 64 bytes). The bound check of the highest dimension exists since its item number is unknown in compilation.

(3) Nested array. We denote a n-dimensional nested array as $\mathsf{T}[X_1]...[X_n]$, where T is one of basic types. At least one of X_1 to X_{n-1} should be empty indicating the corresponding dimension is dynamic. The key difference between a n-dimensional nested array and a n-dimensional dynamic array is that at least one dimension of the lower $n-1$ dimensions of the former can be dynamic whereas the lower $n-1$ dimensions of the latter must be static [3]. Hence, each dimension of a n-dimensional nested array associates an *offset* field and a *num* field to enable dynamic dimension size. Fig. 7 shows the call data layout of a nested array uint[][], which is the first parameter of a function. Let's assume that the actual argument is [[0x1, 0x2], [0x3]] for the ease of explanation. $offset_1$ is the *offset* field of this nested array, and num_1 is the *num* field of its highest dimension. Two items (i.e., [0x1, 0x2] and [0x3]) in the highest dimension are placed after num_1. Because both the two items are dynamic arrays, their *offset* fields will be placed immediately after num_1. In this example, $offset_2$ and num_2 are the *offset* and *num* fields of the item [0x1, 0x2], and $offset_3$ and num_3 are the *offset* and *num* fields of the item [0x3]. 0x1 and 0x2 which are two items of the array [0x1, 0x2] are placed immediately after num_2. Similarly 0x3, the only item of the array [0x3] is placed immediately after num_3.

The accessing pattern of a nested array in the public mode is the same as that in the external mode, and an array item in nested array will be read from the call data to the stack by a CALLDATALOAD on demand. More specifically, two CALLDATALOADs are used to read $offset_1$ and num_1, the item number of the highest dimension. For example, to read the array item 0x3 as an example, three additional CALLDATALOADs are required to read $offset_3$, num_3 and the item 0x3. To access an array item, there is a bound check for each dimension.

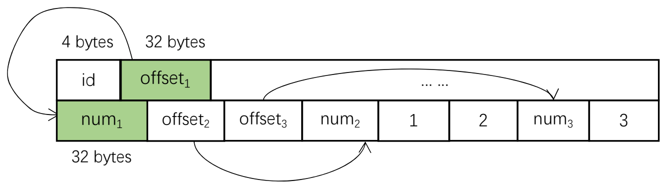

Fig. 7. The call data layout of a nested array uint[][], and the actual argument is [[1, 2], [3]]

3. bytes: It is a dynamic byte sequence whose size is determined at runtime [3]. If a public/external function has a bytes type parameter, it will be extended on the lower-order (right) side with the minimum number of zero to make the length of the extended result be multiple of 32 bytes [3]. For example, if an un-extended bytes is 'abcd', the extended result is shown in Fig. 4. The layout of a bytes is similar to that of a one-dimensional dynamic array, as shown in Fig. 6, except that the *num* field stores the size (in byte) of the bytes before padding. Note that the *num* field of a one-dimensional dynamic array records the number of array items. The extended value is located right after the *num* field.

The accessing patterns of a bytes value are different for public and external functions.

(a) *Public function.* The accessing pattern of a bytes in a public function is the same as the accessing pattern of a one-dimensional dynamic array in a public function except the computation of read size, because every single byte of a bytes is not extended.

(b) *External function.* The accessing pattern of a bytes in the external mode is the same as the accessing pattern of a one-dimensional dynamic array in the external mode, except that accessing a single byte of a bytes does not need the multiplication of 32 because every single byte of a bytes is not extended.

4. string: It is a dynamic Unicode string, whose size is determined at runtime [3]. The call data layout of a string and reading a string from the call data are the same as that of a bytes. We distinguish them based on the observation that a bytes supports reading/writing its individual byte whereas a string does not support such operation.

5. struct: We denote a struct as $(T_1, ..., T_n)$, where n is a constant number, indicating n struct items whose types can be any types in Solidity. The size of a struct can be dynamic or static. For a static struct whose size is known before compilation, the type of each struct item should be one of basic types, static array, or static struct. A dynamic struct whose size is unknown in compilation can host all possible Solidity types. The accessing pattern of a struct in the public mode is the same as that in the external mode. We find that the layout of a static struct is the same as that of all its items as if they were not inside the static struct. For example, Listing 2 is the definition of a struct (uint256, uint256) in Solidity, Listing 3 is the definition of a function with two uint256 parameters in Solidity. Fig. 8 shows the same call data layout of them. Moreover, the bytecode to read an item of a static struct is the same as the bytecode to read the item as if it was not placed inside the struct. Therefore, there is no sufficient hint from the bytecode to distinguish these two different situations.

```
struct StructSample {
  uint256 a;
  uint256 b;
}
function Func(StructSample var1) {}
```

Listing 2. A function taking in a struct with two uint256 items

```
function FuncSample(uint256 a, uint256 b) {}
```

Listing 3. A function taking in two uint256 parameters

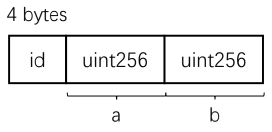

Fig. 8. The layout of a struct containing two uint256 items is the same to that of two individual uint256 parameters

The layout of a dynamic struct has an *offset* field, and all struct items are placed after the *offset* field. Fig. 9 shows the call data layout of a dynamic struct (uint[], uint), which is the first parameter of a function, and we assume that the actual argument is ([0x1, 0x2], 0x3). In this example, $offset_1$ is the *offset* field of this struct. Two items (i.e., [0x1, 0x2] and 0x3) of this struct are placed after the position which $offset_1$ indicated. Because the item [0x1, 0x2] is array, its *offset* field will be placed immediately after the position which $offset_1$ indicated, and the value of item 0x3 will be placed immediately after it. In this example, $offset_2$ and num_1 are the *offset* and *num* fields of item [0x1, 0x2], and items of

[0x1, 0x2] are placed immediately after num_1. For example, to access the struct item 0x3, one CALLDATALOAD is used for reading $offset_1$ and one additional CALLDATALOAD is required to read the item 0x3 directly.

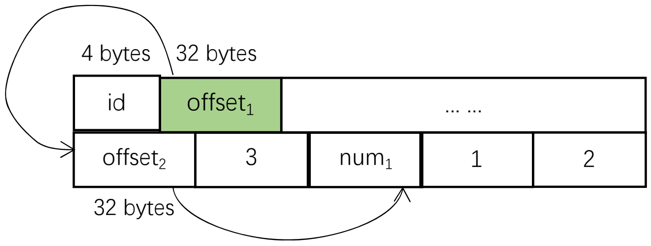

Fig. 9. The call data layout of a dynamic struct (uint[],uint), and the actual argument is ([1, 2], 3)

2.3.2 Vyper

Vyper supports ten parameter types, including bool, int128, uint256, address, bytes32, decimal, fixed-size list, fixed-size byte array, fixed-size string, and struct [33]. The first five types are also supported by Solidity, and the layouts of these five types are the same with the layouts of Solidity types. Different with Solidity, the instructions to read these five types use comparison instructions (e.g., LT) rather than mask instructions (e.g., AND) to ensure that the values of these five types are in the valid ranges. For example, Listing 4 shows the accessing pattern for address parameter in Solidity and Listing 5 shows the accessing pattern for address parameter in Vyper. We also learn that Vyper generates the same bytecode for public and external functions. We describe the remaining five types below.

```
CALLDATALOAD //read the parameter
 ...
PUSH20 0xffffffffffffffffffffffffffffffffffffffff
AND //mask by 0xff...ff
```

Listing 4. Accessing pattern for an address parameter in Solidity

```
PUSH21 0x010000000000000000000000000000000000000000
PUSH1 0x20
MSTORE
 ...
CALLDATALOAD //read the parameter
PUSH1 0x20
MLOAD //load 0x01...00
DUP2
LT //compare to 0x01...00
```

Listing 5. Accessing pattern for an address parameter in Vyper

1. decimal. A decimal is a fixed-point value with a precision of 10 decimal, whose value ranges from -2^{127} and $2^{127}-1$ [33]. We find that the layout and the accessing pattern of a decimal is similar to that of signed integer in Solidity, except that two comparisons rather than a SIGNEXTEND mask are used for ensuring that the value of the decimal is between -2^{127} and $2^{127}-1$. If the decimal value is out of the range, the execution of smart contract will be aborted.

2. Fixed-size list. Fixed-size list records a constant number of items. [33]. We denote a fixed-size list as $T[X_1]...[X_n]$, where T is one of bool, int128, uint256, address, bytes32 and decimal. $X_1, ..., X_n$ are constant numbers known in compilation, and n is the dimension and $n \geq 1$. The layout and accessing pattern of a fixed-size list is the same as that of a static array in the external mode of Solidity. To access a list item, bound checks are used to prevent the overrun of the list if the index is a variable.

3. Fixed-size byte array. A fixed-size byte array is a byte sequence with a maximum length, which is given in the source code. Its real length which is provided at runtime, should not be longer than the maximum size. We denote a fixed-size byte array as bytes[$maxLen$], $maxLen$ is a constant value denoting the maximum length. The layout and accessing pattern for a fixed-size byte array are similar to a bytes in the public mode of Solidity, except that 32 (the size of the num field) + $maxLen$ are read from the call data. That is, the extended bytes of a fixed-size byte array are not read.

4. Fixed-size string. We denote a fixed-size string as string[$maxLen$], where $maxLen$ is a constant value representing the maximum length of the string. The layout and the way to read a fixed-size string from the call data are the same as that of a fixed-size byte array. The difference between the two types lies in that a fixed-size byte array allows accessing its individual byte whereas a fixed-size byte array does not allow it.

5. struct. struct is a container type that can host several variables. We denote a struct as $(T_1, ..., T_n)$, where n is a constant value indicating n struct items and T_i can be one of the following types, bool, int128, uint256, address, bytes32 and decimal [33]. The layout of a struct is the same as that of all its items as if they were not placed inside the struct. The bytecode to read a struct item is also the same as the bytecode to read the item as if it was not inside the struct. Therefore, there is no sufficient hint from the bytecode to distinguish these two different types. Listing 6 shows a function signature which takes in a struct containing two uint256 items, and Listing 7 presents a function taking in two uint256 parameters. The layouts of the struct and two individual uint256 parameters are the same, as shown in Fig. 10.

```
struct StructSample:
   a: uint256
   b: uint256
def Func(var1: StructSample)
```

Listing 6. A function taking in a struct with two uint256 items

```
def FuncSample(a: uint256, b: uint256)
```

Listing 7. A function taking in two uint256 parameters

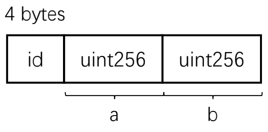

Fig. 10. The layout of a struct containing two uint256 items is the same to that of two individual uint256 parameters

3 Rules for Type Inference

This section first introduces the method to generate rules, and then describes the first four rules. We briefly introduce the remaining 27 rules, and elaborate more on them in Supplementary material C.

3.1 Rules Generation

We propose a systematic approach to generate rules. The basic idea is to first learn the accessing patterns automatically from the bytecode of self-generated smart contracts and then summarize the rules manually, which are used by SigRec to recover function signatures from the bytecode of other smart contracts. Our approach has five steps. Since the first four steps are automated, it can be easily extended to handle new accessing patterns due to new compilers/programming language/obfuscation techniques and new parameter types (discussed in §7).

Step 1. Preparing smart contracts and their bytecode. We develop a tool to automatically construct smart contracts and compile them into bytecode, from which the rules will be learned. We develop smart contracts in Solidity and Vyper since they are the most popular languages for developing Ethereum smart contracts. To ensure the completeness of the derived rules, we consider all parameter types supported in Solidity and Vyper, all major versions of compilers as well as various optimization levels. More specifically, each smart contract contains one public or external function. Since we find that smart contracts handle the different parameters of a function independently, each generated function just has one parameter to ease rule learning. To expose the accessing patterns of parameters, the body of each generated function contains statements to access the parameter, e.g., assignments, arithmetic operations, comparisons, reading one byte from bytes and bytes32, and reading one array item.

We consider all basic types with all possible widths, e.g., uint8, uint16, ..., uint256. Since the upper bound of the array dimension and the upper bound of the size of a static dimension are undocumented, we first identify the patterns with concrete number of dimensions and then generalize the pattern for all possible dimensions. Specifically, we set array dimensions from 1 to 5 and each dimension can be static or dynamic. For each static dimension, we set its size from 1 to 10. The items of an array are set to one of supported basic types with all possible widths (e.g., uint8, uint16, ..., uint256).

Step 2. Collecting the accessing pattern of each parameter type. We conduct data dependence analysis and control dependence analysis to locate the EVM instruction sequence used to access the parameter of each generated function, which are required in subsequent steps. We consider such instruction sequence as an accessing pattern. Specifically, the instructions which are data dependent with parameters are needed in step 4 for collecting the symbolic expressions of variables. Step 5 needs the control dependency relations to infer the structure of some complicated parameter types. For example, TASE infers the loop structure from the EVM instructions that are control dependent on the instruction for reading the call data to discover *n*-dimensional dynamic array.

Step 3. Extracting common accessing patterns. We observe that the access patterns for some parameters types (e.g., uint8 and uint16) are similar, and thus this step extracts common access patterns from the accessing patterns obtained in step 2. We regard an instruction sequence as the *common* accessing pattern of some accessing patterns if it appears in all these accessing patterns. For example, from step 2 we obtain the accessing patterns of uint8, uint16, ..., uint256 in Solidity, and then in this step we try to extract the common accessing pattern from these patterns to facilitate the derivation of the rule for uint with all possible widths. On the contrary, bool, int128, uint256, address, bytes32, decimal of Vyper are basic types, we skip the step for them because these types have fixed widths. Besides, we omit the description about the fixed-size list of Vyper because it is equivalent to the static array of Solidity.

Basic types of Solidity. We use unsigned integers as an example, and the other basic types are handled similarly. We extract the common accessing pattern from the accessing patterns of uint8, uint16, ..., uint256, which is used in step 5 to generate a general rule for inferring the unsigned integers with all possible widths, uint$\langle M \rangle$, $8 \leq M \leq 256, M\%8 == 0$.

One-dimensional static array of Solidity. We extract the common accessing pattern from the accessing patterns of uint8[1], uint8[2], ..., uint8[10]. By retaining the instructions in the common accessing pattern but not in the accessing pattern of uint8, we further obtain the accessing pattern of T[N], $1 \leq N \leq 10$, T is one of the basic types.

Multidimensional static array of Solidity. We first extract the common accessing pattern from the accessing patterns of uint8[N_1][N_2], $1 \leq N_1, N_2 \leq 10$. Then we obtain the accessing pattern of T[N_1][N_2], $1 \leq N_1, N_2 \leq 10$ by retaining the instructions in the common accessing pattern but not in the accessing pattern of uint8. Similarly, we obtain the accessing patterns of T[N_1][N_2][N_3], T[N_1][N_2][N_3][N_4], T[N_1][N_2][N_3][N_4][N_5], $1 \leq N_1, N_2, N_3, N_4, N_5 \leq 10$.

One-dimensional dynamic array of Solidity. We regard the EVM instructions that are in the accessing pattern of uint8[] but not in the accessing pattern of uint8 as the common accessing pattern of T[].

Multidimensional dynamic array of Solidity. We extract the common accessing pattern from accessing patterns of uint8[N_1][], $1 \leq N_1 \leq 10$. By retaining the EVM instructions that appear in such common accessing pattern but not in the accessing pattern of uint8, we obtain the accessing pattern of T[N_1][], $1 \leq N_1 \leq 10$. We also obtain the accessing patterns of T[N_1][N_2][], T[N_1][N_2][N_3][], T[N_1][N_2][N_3][N_4][], $1 \leq N_1, N_2, N_3, N_4 \leq 10$ through the same process.

Nested array of Solidity. We extract the common accessing pattern from the accessing patterns of uint8[][N_1], $1 \leq N_1 \leq 10$ or N_1 is empty. By retaining the EVM instructions that appear in such common accessing pattern but not in the accessing pattern of uint8, we obtain the accessing pattern of T[][N_1], $1 \leq N_1 \leq 10$ or N_1 is empty. We also obtain the accessing pattern of T[N_1][N_2][N_3]][N_4]][N_5], $1 \leq N_1, N_2, N_3, N_4, N_5 \leq 10$ and at least one of N_1 to N_4 is empty through the same process.

Fixed-size byte array of Vyper. We obtain the accessing pattern of bytes[*maxLen*] by extracting the common accessing pattern from the accessing patterns of bytes[1], bytes[2],..., bytes[50].

Fixed-size string of Vyper. We obtain the accessing pattern of string[*maxLen*] by extracting the common accessing pattern from the accessing patterns of string[1], string[2],..., string[50].

struct of Solidity. we extract the accessing pattern of (uint8[]). By retaining the EVM instructions that appear in such accessing pattern but not in the accessing pattern of uint8[], which is extracted above, we obtain the accessing pattern of the struct type of Solidity.

struct of Vyper. we extract the common accessing pattern from accessing patterns of (uint256). By retaining the EVM instructions that appear in such common accessing pattern but not in the accessing pattern of uint256, we obtain the accessing pattern of the struct type of Vyper.

Step 4. Generating symbolic expressions from parameters. To characterize how parameters are handled by EVM instructions, we conduct symbolic execution on the common accessing patterns obtained in step 3 by treating the call data as symbols, and collect the symbolic expressions for each variable. Step 5 needs symbolic expressions to summarize rules.

Step 5. Summarizing rules. We summarize the rules according to the common accessing patterns obtained in step 3 and the symbolic expressions collected in step 4, and then organize them into a decision tree (Fig. 13) for fast rule checking. To determine the rules for arrays, we investigate how the accessing patterns change when its dimension increases from 1 to 5 and the number of items per-dimension increases from 1 to 10. Eventually, we generate 18 rules and divide them into three categories for CALLDATALOAD (§3.2), CALLDATACOPY (§3.3), and other instructions (§3.4), respectively.

3.2 Rules for CALLDATALOAD

A CALLDATALOAD can read 3 kinds of data from the call data (§2): (1) the value of a parameter; (2) the *offset* field of a parameter; (3) the *num* field of a parameter.

R1: R1 is used to infer a dynamic array/bytes/string. It holds if two CALLDATALOADs (termed by CALLDATALOAD_1 and CALLDATALOAD_2) satisfy that $x = \text{CALLDATALOAD}_1(loc) \wedge y = \text{CALLDATALOAD}_2(x + 0\text{x}4)$, where $x = \text{CALLDATALOAD}_1(loc)$ means that a 32-bytes value is read from the *loc* byte of the call data into x by a CALLDATALOAD. Actually, x is the *offset* field of a dynamic array/bytes/string parameter. CALLDATALOAD_2 reads the *num* field of the parameter, because the location of the *num* field is the value of the *offset* field plus the length of the function id (i.e., 4 bytes).

R2: R2 is used to infer an n-dimensional ($n > 1$) dynamic array in an external function. It holds if three requirements, $v1$, $v2$, $v3$ are satisfied. Let $exp(loc)$ represent the symbolic expression of the location *loc*, which describes how *loc* is computed from symbols. Let $exp(p) \diamond q$ denote that the symbolic expression of *p* (e.g., $x + y \times 5$) contains the symbolic expression *q* (e.g., $y \times 5$). $v1$ is defined as $exp(loc) \diamond (\textit{offset}+)$, meaning that the read location is computed by adding the value of the *offset* field, because array items are placed after the *num* field, which is pointed by the *offset* field. Hence, if a CALLDATALOAD reads an item from a dynamic array in the external model, $v1$ is satisfied.

$v2$ is defined as $exp(loc) \diamond (32\times)$, meaning that the symbolic expression of *loc* contains the multiplication of 32. If a CALLDATALOAD reads an item from a dynamic array parameter in an external function, $v2$ is satisfied, because each array item is extended to 32 bytes so that the multiplication of 32 is needed to access an array item.

Before introducing $v3$, we let $num_n = \text{CALLDATALOAD}(num)$ indicate that a CALLDATALOAD reads the value of the *num* field and assigns it to num_n. $num_{n-1}, ..., num_1$ are constant numbers, and $\text{LT}_j(i_j, num_j)$ means that an LT instruction compares a number i_j with a number num_j. Let $ins_i \lhd ins_j$ indicate that the instruction ins_j is control-flow dependent on the instruction ins_i. $v3$ is defined as $\text{LT}_n(i_n, num_n) \lhd \text{LT}_{n-1}(i_{n-1}, num_{n-1}) \lhd ... \lhd \text{LT}_1(i_1, num_1) \lhd \text{CALLDATALOAD}(loc)$. $v3$ is satisfied if the CALLDATALOAD for reading the array item is within a nested loop. Hence, if a $\text{CALLDATALOAD}(loc)$ reads an item from a dynamic array in an external function, $v3$ is satisfied, because n bound checks (i.e., n LTs) for preventing array overrun are located between the two CALLDATALOADs. Hence, if $v1$, $v2$, and $v3$ are all fulfilled, the dynamic array is n-dimensional, and $num_{n-1}, ..., num_1$ are the sizes of the lower $n-1$ dimensions, respectively. We give an example in Supplementary material D to explain this rule.

R3: R3 is used to infer an n-dimensional ($n > 1$) static array in an external function. It depends on two requirements $v1$ and $v2$. $v1$ is defined as $\neg(exp(loc) \diamond (\textit{offset}))$, where "$\neg$" means negation. $v1$ means that the read location is not computed from the *offset* field. If a CALLDATALOAD reads an item from a static array in an external function, $v1$ holds because the layout of a static array does not have an *offset* field (Fig. 5). $v2$ is defined as $\text{LT}_n(i_n, num_n) \lhd \text{LT}_{n-1}(i_{n-1}, num_{n-1}) \lhd ... \lhd \text{LT}_1(i_1, num_1) \lhd \text{CALLDATALOAD}(loc)$. $v2$ is the same as $v3$ of R2. If a $\text{CALLDATALOAD}(loc)$ reads an item from an n-dimensional static array parameter in an external function, $v2$ holds because there are n bound checks for each dimension in a nested loop to prevent array overrun before reading the array item. Hence, if $v1$ and $v2$ are satisfied, the static array is n-dimensional, and the item numbers from the highest dimension to the lowest dimension are $num_n, ..., num_1$, which are all constants since the item number of each dimension in a static array is known in compilation. Supplementary material D uses an example to explain this rule due to page limit.

R4: x is regarded as a uint256, if R1, R2 and R3 are not fulfilled. R4 means that without sufficient hints we just know that the length of x is 32 bytes and thus regard a 32-bytes parameter as a uint256. We will refine it to a specific type after using other rules to get more hints.

R19 is used for inferring a struct nested array parameter. R21 and R22 are used for inferring a struct parameter (R21) and a nested array parameter (R22). R24 and R25 are used for inferring a fixed-size list parameter (R24) and a uint256 parameter (R25) after R20 is satisfied.

3.3 Rules for CALLDATACOPY

A CALLDATACOPY reads the value of a parameter(§2). R5 – R10 and R23 are used for inferring a one-dimensional dynamic array/bytes/string parameter in a public function (R5), a one-dimensional static array in a public function (R6), a one-dimensional dynamic array in a public function (R7), a bytes/string in a public function (R8), an $(n+1)$-dimensional ($n > 0$) static array in a public function (R9), an $(n+1)$-dimensional ($n > 0$) dynamic array in a public function (R10) and a fixed-size byte array and string in Vyper (R23).

3.4 Rules for Other Instructions

R11 – R18 will be applied after R4 is satisfied (Fig. 13). By leveraging them, we can refine a uint256 parameter into a $\text{uint}\langle 256 - 8 \times x\rangle, 0 < x < 32$ (R11), a $\text{bytes}\langle 32 - x\rangle, 0 < x < 32$ (R12), an $\text{int}\langle (x+1) \times 8\rangle, 0 \leq x < 31$ (R13), a bool (R14), an int256 (R15), an address (R16), a bytes (R17), and a bytes32 (R18). R20 is used to distinguish Vyper bytecode from Solidity bytecode. R26 is used for inferring a fixed-size byte array parameter in Vyper after R23 is satisfied. R27 – R31 will be applied after R25 is satisfied (Fig. 13). By leveraging them, we can refine a uint256 parameter into an address (R27), an int128 (R28), a decimal (R29), a bool (R30), a bytes32 (R31) in Vyper.

Rules for CALLDATALOAD

$$\frac{x\text{=CALLDATALOAD}_1(loc) \qquad y\text{=CALLDATALOAD}_2(x+4)}{\text{Solidity unknown} \blacktriangleright \text{dynamic array/bytes/string}} \quad \textbf{(R1)}$$

$$\frac{\begin{array}{c} exp(loc)\diamond(offset+) \qquad exp(loc)\diamond(32\times) \qquad \text{LT}_n(i_n,num_n)\triangleleft\text{LT}_{n-1}(i_{n-1},num_{n-1})\triangleleft\ldots\triangleleft\text{LT}_1(i_1,num_1)\triangleleft\text{CALLDATALOAD}(loc) \\ num_n\text{=CALLDATALOAD}(num) \qquad num_{n-1},\ldots,num_1\text{: Constant}\end{array}}{\text{dynamic array/bytes/string} \blacktriangleright \text{dynamic array in external mode}} \quad \textbf{(R2)}$$

$$\frac{\begin{array}{c} \neg(exp(loc)\diamond(offset)) \qquad \text{LT}_n(i_n,num_n)\triangleleft\text{LT}_{n-1}(i_{n-1},num_{n-1})\triangleleft\ldots\triangleleft\text{LT}_1(i_1,num_1)\triangleleft\text{CALLDATALOAD}(loc) \\ num_n,\ldots,num_1\text{: Constant}\end{array}}{\text{Solidity unknown} \blacktriangleright \text{static array in external mode}} \quad \textbf{(R3)}$$

$$\frac{x\text{=CALLDATALOAD}(loc)}{\text{Solidity unknown} \blacktriangleright \text{uint256}} \quad \textbf{(R4)}$$

$$\frac{\begin{array}{c} offset_1\text{=CALLDATALOAD}_1(loc_1)\triangleleft offset_2\text{=CALLDATALOAD}_2(loc_2)\triangleleft\ldots\triangleleft offset_n\text{=CALLDATALOAD}_n(loc_n) \qquad n>2 \\ (loc_1\text{: Constant})\wedge(exp(loc_2)\diamond(offset_1))\wedge\ldots\wedge(exp(loc_n)\diamond(offset_{n-1}))\end{array}}{\text{Solidity unknown} \blacktriangleright \text{struct/nested array}} \quad \textbf{(R19)}$$

$$\frac{\begin{array}{c} offset_1\text{=CALLDATALOAD}(loc)\triangleleft\nexists LT(i,\text{CALLDATALOAD}(offset_1+0x4))\triangleleft\text{CALLDATALOAD}(offset_1+0x4+0x32\times i) \\ loc\text{: Constant}\end{array}}{\text{struct/nested array} \blacktriangleright \text{struct}} \quad \textbf{(R21)}$$

$$\frac{\begin{array}{c} \text{LT}(i_n,num_n)\triangleleft\text{LT}(i_{n-1},num_{n-1})\triangleleft\ldots\triangleleft\text{LT}(i_1,num_1)\triangleleft\text{CALLDATALOAD}_{n+1}(loc) \\ exp(loc)\diamond(offset_n) \qquad offset_n \text{ is defined in R19} \\ num_x\text{:Constant/Variable} \qquad \text{at least one of } num_1\ldots num_{n-1} \text{ should be variable}\end{array}}{\text{struct/nested array} \blacktriangleright \text{nested array}} \quad \textbf{(R22)}$$

$$\frac{\begin{array}{c} \neg(exp(loc)\diamond(offset)) \qquad \text{LT}_n(i_n,num_n)\triangleleft\text{LT}_{n-1}(i_{n-1},num_{n-1})\triangleleft\ldots\triangleleft\text{LT}_1(i_1,num_1)\triangleleft\text{CALLDATALOAD}(loc) \\ num_n,\ldots,num_1\text{: Constant}\end{array}}{\text{Vyper unknown} \blacktriangleright \text{fixed-size list}} \quad \textbf{(R24)}$$

$$\frac{x\text{=CALLDATALOAD}(loc)}{\text{Vyper unknown} \blacktriangleright \text{uint256}} \quad \textbf{(R25)}$$

Rules for CALLDATACOPY

$$\frac{\nexists\text{LT}\triangleleft\text{CALLDATACOPY}(offset_m,x+36,len) \qquad x \text{ is defined in R1}}{\text{dynamic array/bytes/string} \blacktriangleright \text{one-dimensional dynamic array/bytes/string in public mode}} \quad \textbf{(R5)}$$

$$\frac{\nexists\text{LT}\triangleleft\text{CALLDATACOPY}(offset_m,offset_c,len) \qquad offset_c,len\text{: Constant}}{\text{Solidity unknown} \blacktriangleright \text{one-dimensional static array in the public mode}} \quad \textbf{(R6)}$$

$$\frac{len\text{==}32\times y \qquad len \text{ is defined in R5} \qquad y \text{ is defined in R1}}{\text{one-dimensional dynamic array/bytes/string} \blacktriangleright \text{one-dimensional dynamic array in public mode}} \quad \textbf{(R7)}$$

$$\frac{len\text{==}32\times\lceil y/32\rceil \qquad len \text{ is defined in R5} \qquad y \text{ is defined in R1}}{\text{one-dimensional dynamic array/bytes/string} \blacktriangleright \text{bytes/string in public mode}} \quad \textbf{(R8)}$$

$$\frac{\begin{array}{c} \text{LT}_n(i_n,num_n)\triangleleft\text{LT}_{n-1}(i_{n-1},num_{n-1})\triangleleft\ldots\triangleleft\text{LT}_1(i_1,num_1)\triangleleft\text{CALLDATACOPY}(offset_m,offset_c,len) \\ num_n,\ldots,num_1,len\text{: Constant}\end{array}}{\text{Solidity unknown} \blacktriangleright \text{(n+1)-dimensional static array in public mode}} \quad \textbf{(R9)}$$

$$\frac{\begin{array}{c} \text{LT}_n(i_n,y)\triangleleft\text{LT}_{n-1}(i_{n-1},num_{n-1})\triangleleft\ldots\triangleleft\text{LT}_1(i_1,num_1)\triangleleft\text{CALLDATACOPY}(offset_m,offset_c,len) \qquad y \text{ is defined in R1} \\ num_{n-1},\ldots,num_1\text{: Constant}\end{array}}{\text{dynamic array/bytes/string} \blacktriangleright (n+1)\text{-dimensional dynamic array in public mode}} \quad \textbf{(R10)}$$

$$\frac{x\text{=CALLDATALOAD}(loc) \qquad \text{CALLDATACOPY}(offset_m,offset_c,len) \qquad len\text{:Constant} \qquad exp(offset_c)\diamond(x)}{\text{Vyper unknown} \blacktriangleright \text{fixed-size byte array/string}} \quad \textbf{(R23)}$$

Rules for Other Instructions

$$\frac{\text{AND}(op_1,op_2) \qquad op_1\text{:uint256} \qquad op_2\text{:Constant with } x \text{ leading zero-bytes}}{\text{uint256} \blacktriangleright \text{uint}\langle\text{M}\rangle} \quad \textbf{(R11)}$$

$$\frac{\text{AND}(op_1,op_2) \qquad op_1\text{:uint256} \qquad op_2\text{:Constant with } x \text{ trailing zero-bytes}}{\text{uint256} \blacktriangleright \text{bytes}\langle\text{M}\rangle} \quad \textbf{(R12)}$$

$$\frac{\text{SIGEXTEND}(op,x) \qquad op\text{:uint256} \qquad x\text{:Constant}}{\text{uint256} \blacktriangleright \text{int}\langle\text{M}\rangle} \quad \textbf{(R13)} \qquad \frac{x\text{=ISZERO}(op) \qquad y\text{=ISZERO}(x) \qquad op\text{:uint256}}{\text{uint256} \blacktriangleright \text{bool}} \quad \textbf{(R14)}$$

$$\frac{\text{SDIV/SMOD/SLT/SGT}(op_1,op_2) \qquad op_1\text{:uint256}}{\text{uint256} \blacktriangleright \text{int256}} \quad \textbf{(R15)} \qquad \frac{op_1\text{:uint160} \qquad op_1 \text{ is not involved in any MATH}(op)}{\text{uint256} \blacktriangleright \text{address}} \quad \textbf{(R16)}$$

$$\frac{\text{BYTE/MSTORE8}(op) \qquad op\text{:string/bytes}}{\text{bytes/string} \blacktriangleright \text{bytes}} \quad \textbf{(R17)} \qquad \frac{\text{BYTE}(op) \qquad op\text{:uint256}}{\text{uint256} \blacktriangleright \text{bytes32}} \quad \textbf{(R18)}$$

$$\frac{x\text{=CALLDATALOAD}(0) \qquad \text{MSTORE}(offset_m,x)}{\text{unknown} \blacktriangleright \text{Vyper unknown}} \quad \textbf{(R20)} \qquad \frac{\text{BYTE/MSTORE8}(op) \qquad op\text{:fixed-size byte array/string}}{\text{fixed-size byte array/string} \blacktriangleright \text{fixed-size byte array}} \quad \textbf{(R26)}$$

$$\frac{\text{LT}(op_1,op_2) \qquad op_1\text{:uint256} \qquad op_2\text{:}2^{160}}{\text{uint256} \blacktriangleright \text{address}} \quad \textbf{(R27)}$$

$$\frac{\text{SLT}(op_1,op_2) \qquad \text{SGT}(op_1,op_3) \qquad op_1\text{:uint256} \qquad op_2\text{:}2^{127}-1 \qquad op_3\text{:}-2^{127}}{\text{uint256} \blacktriangleright \text{int128}} \quad \textbf{(R28)}$$

$$\frac{\text{SLT}(op_1,op_2) \qquad \text{SGT}(op_1,op_3) \qquad op_1\text{:uint256} \qquad op_2\text{:}2^{127}-1 \qquad op_3\text{:}-2^{127}}{\text{uint256} \blacktriangleright \text{decimal}} \quad \textbf{(R29)}$$

$$\frac{\text{LT}(op_1,op_2) \qquad op_1\text{:uint256} \qquad op_2\text{:}2}{\text{uint256} \blacktriangleright \text{bool}} \quad \textbf{(R30)} \qquad \frac{\text{BYTE/MSTORE8}(op) \qquad op\text{:string/bytes}}{\text{uint256} \blacktriangleright \text{bytes32}} \quad \textbf{(R31)}$$

Fig. 11. Rules

4 Design and Implementation of SigRec

4.1 Overview

Fig. 12 shows the architecture of `SigRec` which takes in the *runtime* bytecode of a smart contract and outputs the function signatures of all public/external functions in it. `SigRec` first disassembles the bytecode using Geth disassembler [34] and recognizes basic blocks from them, then extracts function ids from the bytecode. Technical details are presented in Supplementary material E. After that, it uses TASE to infer the types of all parameters (§4.2). Finally, it outputs the function ids as well as the list of parameter types. The reasons for using TASE rather than conventional SE and other methods are explained in Supplementary material F.

4.2 TASE: Type-aware Symbolic Execution

Being the core of `SigRec`, TASE has four steps. First, it conducts coarse-grained type inference to recognize Solidity and Vyper bytecode, and then it recognizes struct, arrays, bytes, strings and basic types of Solidity, and fixed-size lists, fixed-size byte arrays, fixed-size strings and basic types of

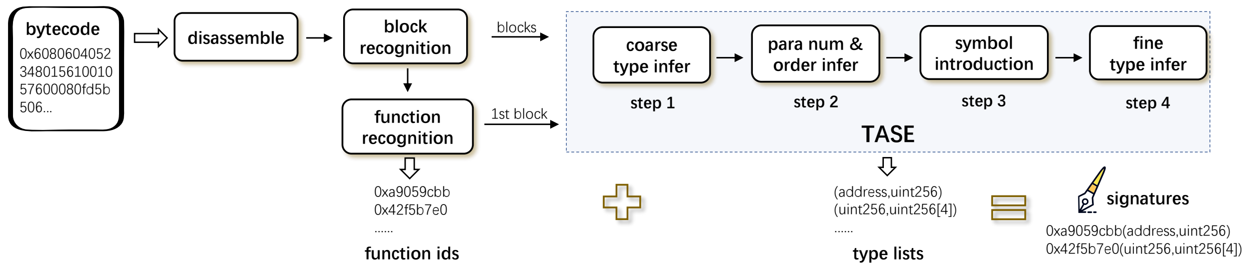

Fig. 12. Architecture of `SigRec`

Vyper. This step only determines whether a parameter is a basic type instead of deciding the specific basic type, and this step does not infer the type of array items, the type of struct items, and this step does not distinguish a bytes from a string. Second, it infers the number and the order of parameters. Third, by introducing parameter-related symbols when reading arguments from the call data, TASE determines whether an instruction operates on an argument and which argument is operated if that is the case. Finally, it conducts fine-grained type inference to distinguish different basic types for Solidity and Vyper, recognizes the item types of arrays, nested arrays, and each item type of struct for Solidity and the item types of fixed-size list for Vyper, and refines the types (e.g., from uint256 to bytes32) for Solidity and Vyper.

TASE treats the call data as symbols and maintains the symbolic expressions of all variables depending on the call data in order to apply the rules. TASE enhances conventional symbolic execution (CSE) with the ability to infer parameter types. It conducts symbolic execution (SE) to statically explores the paths of the smart contract and stops if the jump target is determined by inputs (e.g., function parameters), because inputs are unknown in static analysis. This restriction is not a big problem in practice since we only find 5 smart contracts deployed in Ethereum containing such kind of jump instructions, and TASE usually does not need to analyze the code deep inside smart contracts because parameters are usually handled near the entry point of each function. TASE treats each value read from the environment as a free symbol, because `SigRec` focuses on how a smart contract processes the parameters rather than its program logic. Experimental results show that these two restrictions do not affect `SigRec`'s accuracy (§5.2). After describing the details of TASE, we present the differences between TASE and CSE, as well as the rationale of using TASE instead of other approaches.

Step 1. Coarse-grained type inference.

At this step, TASE recognizes struct, arrays, bytes, strings and basic types in Solidity, and fixed-size lists, fixed-size byte arrays, fixed-size strings and basic types in Vyper according to the rules R1 – R10 and R19 – R25. This step only determines whether a parameter is a basic type instead of deciding the specific basic type. This step does not infer the type of array items, the type of struct items, and this step does not distinguish a bytes from a string. Instead, these tasks are accomplished by step 4.

Fig. 13 shows the decision tree used by `SigRec` to determine the type of a parameter. A rectangle represents one or more types and an edge indicates applying one or more rules. The root rectangle represents the unknown type. Two colors are used to differentiate between a public function and an external function. The ¬ indicates that the requirements of the corresponding rule are not satisfied. `SigRec` infers the type of a parameter if all the rules on a path from the root rectangle to a leaf node are satisfied.

We use the following example to explain the derivation process. `SigRec` regards a parameter as a bytes in a public function if R1, R5, R8, and R17 are fulfilled in order. More precisely, if R1 holds, the parameter should be a dynamic array/bytes/string, because the smart contract uses two consecutive CALLDATALOADs to read the *offset* field and the *num* field. Since R5 is also satisfied, the type should be a one-dimensional dynamic array or a bytes or a string in a public function, because exactly one CALLDATACOPY is used to copy the parameter. Then, R8 refines the type to a bytes or a string in a public function, because a bytes/string is extended to the length of a multiple of 32 bytes. Finally, R17 further refines the type to a bytes in the public mode, because the smart contract accesses a single byte of the parameter.

Step 2. Determining the number and order of parameters.

TASE infers the number of parameters by counting the number of rules R1, R3, R4, R6, R9, R21, R22, R23, R24, R25 which are used in coarse-grained type inference. Specifically, by counting the applied number of R1, TASE knows the number of dynamic arrays/bytes/strings for Solidity (termed by $n1$) since two consecutive CALLDATALOADs are used before reading a dynamic array/bytes/string. By counting the applied number of R3, R6 and R9, TASE knows the number of static arrays for Solidity (termed by $n2$). Since R4 regards all basic types as uint256, TASE knows the number of basic types for Solidity (termed by $n3$) by counting the applied number of R4. By counting the applied number of R21 and R22, TASE knows the number of nested array and struct for Solidity (termed by $n4$). By counting the applied number of R23 and R24, TASE knows the number of fixed-size list, byte array and string for Vyper (termed by $n5$). By counting the applied number of R25, TASE knows the number of basic types for Vyper (termed by $n6$). Therefore, TASE knows the number of parameters, which is $n1+n2+n3+n4+n5+n6$.

Then, TASE determines the order of parameters according to the locations of the corresponding arguments in the call data in ascending order. For example, if an argument x locates before an argument y in the call data, x is on the left-hand side of y in the parameter list. The location of a basic type is given by the operand of the CALLDATALOAD. The location of a dynamic array/bytes/string is indicated by the location of its *offset* field, which is the operand of the first CALLDATALOAD instruction in the two consecutive CALLDATALOAD instructions (R1, §3.2). The location of a static array in a public function is given in the operand of the first

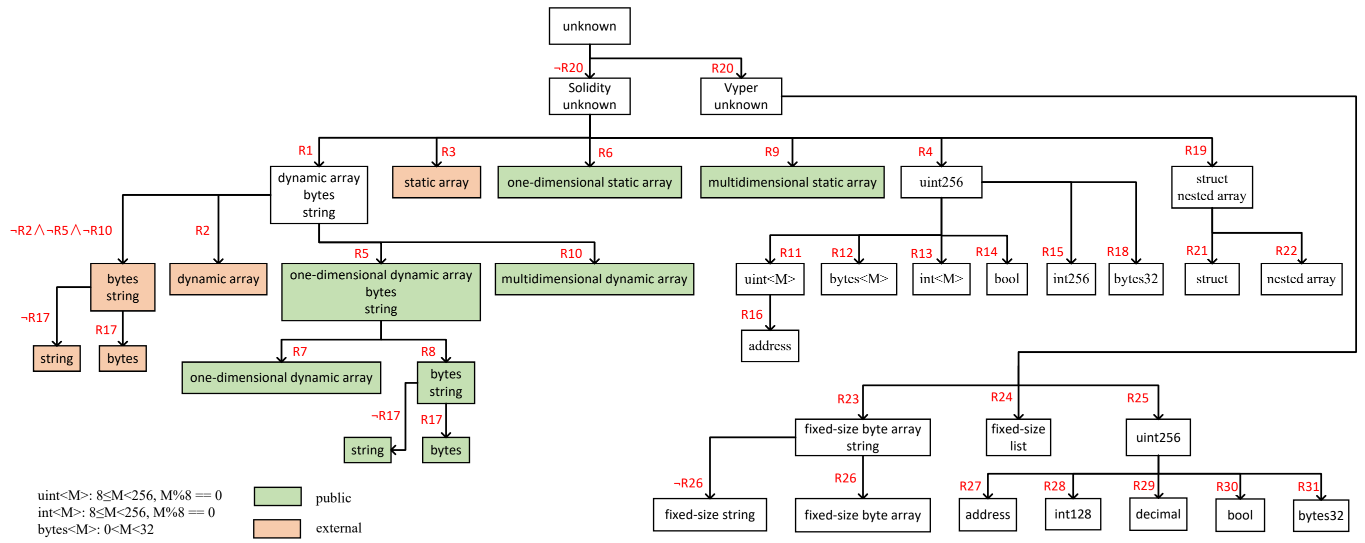

Fig. 13. Hierarchy of rules applied by `SigRec`

CALLDATACOPY because it copies the inner-most dimension. The start location of a static array in an external function cannot be directly obtained because it is not involved in reading array items. Instead, a CALLDATALOAD is used to read an array item whose location is a constant number determined during compilation. Hence, we get the start location indirectly, which is after the end of the parameter immediately before the static array. If the static array is the first parameter, the start location is 0x4 since the function id precedes it.

Step 3. Introducing parameter-related symbols. When a smart contract reads arguments, TASE introduces parameter-related symbols by marking all bytes of an argument with the same symbol. After executing a CALLDATALOAD and a CALLDATACOPY, we mark the corresponding stack item and the memory region with symbols. Besides, we will copy the symbol of the memory region to the top item of the stack after executing an MLOAD if it reads from the memory region that stores arguments. Fig. 14 shows an example of marking parameter-related symbols. The public function, *fun*(), takes in a uint256 parameter x and a uint256[3] parameter y, and assigns a local variable z as $y[0]$. Before the assignment, x and y are read from the call data to the stack top and the memory, respectively. Since x and y are arguments, we mark the stack top and the corresponding memory region as symbols $arg1$ and $arg2$, respectively. The assignment copies the first item of y from the memory to the stack top, and therefore we mark stack top with the same symbol of y (i.e., $arg2$).

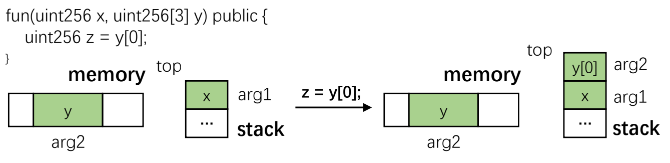
```
fun(uint256 x, uint256[3] y) public {
    uint256 z = y[0];
}
```

Fig. 14. Marking parameter-related symbols

Step 4. Fine-grained type inference.

Using R11–R18 and R26–R31 according to Fig. 13, this step distinguishes basic types for Solidity and Vyper, infers the type of an array item and nested array item for Solidity, infers each item type of struct for Solidity, and infers the type of a fixed-size list item for Vyper, and refines the type of a parameter to a specific one for Solidity and Vyper. Since TASE knows the structure of an array in step 1, it further knows the type of an array item in this step and then determines the type of the array. For example, given a public function with a uint8[] parameter, TASE knows that the parameter is a one-dimensional dynamic array in a public function by applying R1, R5 and R7 in order in step 1. Then, using R11 in this step, TASE learns that the type of an array item is uint8, and thus `SigRec` can recover the correct parameter type.

An example to illustrate the four steps. Listing 8 shows a public function with two parameters: *values* and *to*, whose types are uint8[] and address, respectively. Line 2 reads the first item of *values* and *to*. Listing 9 shows the corresponding EVM bytecode. For the ease of presentation, we just keep the EVM instructions that are needed by TASE to infer parameter types.

```
1 function test(uint8[] values, address to) public {
2   to.send(values[0]);
3 }
```
Listing 8. An example to explain the process of TASE

```
1  CALLDATALOAD //read offset
2  ......
3  CALLDATALOAD //read num
4  ......
5  CALLDATACOPY //read values
6  ......
7  CALLDATALOAD //read to
8  PUSH20 0xff...ff //20byte 0xFF
9  AND //mask
10 ......
11 MLOAD //read values[0]
12 PUSH1 0xff
13 AND //mask
```
Listing 9. The bytecode code of this example

Step 1: Coarse-grained type inference. In Listing 9, Line 1 reads 32 bytes, termed by x, from the 4th byte of the call data, and Line 3 reads 32 bytes, termed by y, from the $(x+4)$-th byte of the call data. Since these two instructions satisfy R1, this parameter is a dynamic array/bytes/string. Since Line 5 reads $32 \times y$ bytes from the $(36+x)$-th byte of the call data to memory from the 160th byte without using a loop, R5 is fulfilled and thus this parameter is refined to one-dimensional dynamic array/bytes/string. `SigRec` further confirms that this parameter is a one-dimensional dynamic array because R7 is satisfied. Specially, the read length is the multiplication of 32 bytes (i.e., the length of each array item) and the item number (i.e., y). As Line 7 reads 32 bytes, termed by z, from the 36th byte of the call data to the stack, R4 is fulfilled and thus z is a basic-type parameter.

Step 2: Determining the number and order of parameters. Line 3 reads a one-dimensional dynamic array, and its *offset* field is located at the 4th byte of the call data. Line 7 reads a basic-type parameter located at the 36th byte of the call data. Since no other parameters are read in this example, this function takes in two parameters. Moreover, the one-dimensional dynamic array is the first parameter and the basic-type parameter is the second parameter, because the *offset* field of the array precedes the basic-type parameter.

Step 3: Introducing parameter-related symbols. Since Line 5 reads the first parameter to the memory, `SigRec` marks the corresponding memory region with the symbol $arg1$. Similarly, since Line 7 reads the second parameter to the stack, `SigRec` marks the stack's top item with the symbol $arg2$.

Step 4: Fine-grained type inference. Since Line 9 masks the stack top, which is marked with $arg2$, with 20-bytes 0xFF, R11 is satisfied and thus the second parameter should be a uint160. Since the second parameter is not involved in any mathematics operations, R16 is held, and thus `SigRec` refines the type of the second parameter to address. Line 11 reads 32 bytes from the 160th byte of the memory to stack top. Since such memory region is marked with $arg1$ in step 3, `SigRec` marks stack top with $arg1$. Since the first parameter is an array, the stack top stores an array item. As Line 13 masks the stack top with 0xFF, R11 is satisfied and thus we learn that the type of the array item is uint8. Eventually, TASE infers that the type list is "uint8[], address", the same as the source code.

5 Evaluation

We implement `SigRec` in 8,327 lines of Python code and conduct extensive experiments to evaluate it by answering five research questions. **RQ1**: How is the accuracy of `SigRec` in recovering function signatures (§5.2)? **RQ2:** Will `SigRec` be affected by different compiler versions and optimizations (§5.3)? **RQ3**: How much time is required by `SigRec` to recover function signatures (§5.4)? **RQ4**: How frequently is each rule used for recovering function signatures (§5.5)? **RQ5**: Is `SigRec` superior to existing tools (§5.6)?

5.1 Data Collection

To evaluate the accuracy of `SigRec`, we download the source code of all open-source smart contracts which were deployed before Jan. 6, 2021 from Etherscan because the ground-truth (i.e., function signatures) can be obtained from the source code. We collect 119,404 unique open-source smart contracts including 119,126 Solidity contracts and 278 Vyper contracts. 210,869 unique public/external function signatures are found in Solidity open-source smart contracts and 1,076 unique function signatures are found in Vyper open-source smart contracts. To evaluate the efficiency of `SigRec` and the usefulness of heuristic rules, we collect the bytecode of all deployed smart contracts by instrumenting an Ethereum full node as suggested by [35], [36]. Eventually, we download 11,600,000 blocks (the last block was mined on Jan. 6, 2021) and obtain the bytecode of 37,009,570 smart contracts. 368,679 out of them are unique. There are 47,329,149 public/external functions in all smart contracts with 383,522 unique function signatures. Note that these 37,009,570 smart contracts include all open-source smart contracts.

5.2 RQ1: How is the accuracy of `SigRec`?

We evaluate the accuracy of `SigRec` with all 210,869 and 1,076 unique function signatures in Solidity and Vyper open-source smart contracts with ground-truth, respectively. A recovered function signature is *correct*, if and only if the recovered function id, the number and the order of parameters, and the types of all parameters are the same as the ground-truth. The *accuracy* is the proportion of correctly recovered function signatures to the total number of function signatures. `SigRec` correctly recovers 208,218 and 1,052 function signatures for Solidity and Vyper, and hence its accuracy is 98.738% ($(208,218+1,052)/(210,869+1,076)$). More specifically, `SigRec` correctly recovers 208,218 from all function signatures in Solidity smart contracts, and thus its accuracy for Solidity contracts is 98.743% ($208,218/210,869$). Besides, `SigRec` correctly recovers 1,052 from all 1,076 function signatures in Vyper smart contracts, and thus its accuracy for Vyper contracts is 97.770% ($1,052/1,076$).

By manually investigate 2,651 incorrect function signatures in Solidity and 24 incorrect function signatures in Vyper, we reveal five cases of inaccuracies as follows. The value in <> is the number of inaccurately recovered function signatures in each case. Please note that one incorrect function signature may be affected by multiple cases. We discuss how to further increase the accuracy of `SigRec` in §7.

Case 1 $<498>$**:** These smart contracts read the parameters that are not declared in function signatures by inline assembly. Listing 10 shows a practical case. The function start() has no parameters in declaration (Line 1), but it reads two parameters *foo* (Line 7) and *bar* (Line 8) using inline assembly. `SigRec` discovers these two parameters read by inline assembly, because it infers parameters by investigating how parameters are used.

```
1  function start() auth note{
2    stopped=false;
3  }
4  modifier note{
5  ......
6    assembly{
7      foo:=calldataload(4)
8      bar:=calldataload(36)
9    }
10 ......}
```

Listing 10. An inaccurately recovered function signature in case 1

Case 2 $<387>$**:** These smart contracts forcibly convert the type of a parameter before using it. The recovered types may be more useful than those declared in function signatures, and the smart contract uses the parameter as the converted type. Listing 11 shows a practical case. The function setGenOStat() has one parameter whose type is declared as uint256[6] (Line 1). `SigRec` recovers the type of *_gen0Stat* as uint8[6], because the high-order 31 bytes of uint256 are discarded by type conversion (Lines 4 – 9).

```
1  function setGen0Stat(uint256[6] _gen0Stat) public onlyCOO
2  {
```

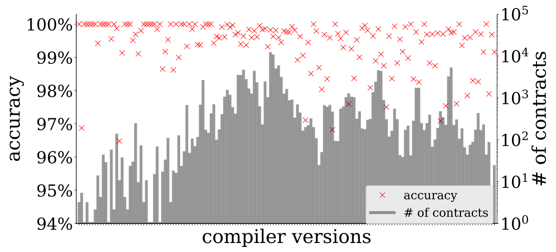

Fig. 15. Accuracies of `SigRec` for various Solidity compiler versions

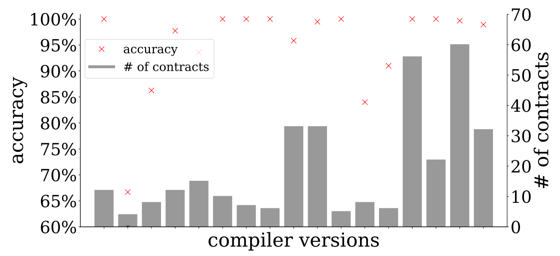

Fig. 16. Accuracies of `SigRec` for Vyper compiler versions

```
    gen0State=Gen0Stat({
    retiredAge:uint8(_gen0Stat[0]),
    maxSpeed:uint8(_gen0Stat[1]),
    maxStamina:uint8(_gen0Stat[2]),
    maxStart:uint8(_gen0Stat[3]),
    maxBurst:uint8(_gen0Stat[4]),
    maxTemperament:uint8(_gen0Stat[5])});
}
```

Listing 11. An inaccurately recovered function signature in case 2

Case 3 $< 314 >$: These smart contracts have unused arrays/strings/bytes parameters in external functions. Since `SigRec` recognizes such types of parameters by checking how they are used, it cannot recognize the unused parameters. Missing such parameters may not be an important issue because they are not used. It is worth noting that `SigRec` can recognize all parameters in public functions and the basic-type parameters in the external functions, because such parameters will be read into the stack or the memory no matter whether or not they will be used.

Case 4 $< 602 >$: Since `SigRec` recovers the type of each parameter with the *storage* modifier as uint256, its output may be incorrect if the storage variable referred by the parameter is not a uint256 integer.

Case 5 $< 1,123 >$: Our rules cannot handle a few scenarios. First, `SigRec` cannot recognize static arrays in external functions, if the smart contract is compiled with optimization and the index of the array item being accessed is a constant number, because there are no bound checks which are necessary for inferring array structure (R2 and R3, §3.2). Second, `SigRec` cannot distinguish some types if it cannot obtain sufficient clues. For example, we cannot distinguish a bytes from a string if the smart contract does not access a single byte of the bytes value since R17 exploits the fact that a bytes allows accessing its individual byte but a string does not support such functionality. As another example, we cannot distinguish a struct whose items are all basic types from the same items which are not placed in a struct, due to the description about struct in §2.3.1 that the call data layout and the accessing pattern of a static struct are the same to that of the same items which are not placed in a struct.

Answer: *The accuracy of `SigRec` is 98.7% for Solidity and 97.8% for Vyper.*

5.3 RQ2: Will `SigRec` be affected by different compiler versions and optimizations?

We evaluate `SigRec` using the open-source smart contracts that are compiled by different versions of compilers w/o optimization. Since Etherscan lists the compiler version and whether or not a smart contract is optimized as well as the optimization level if any for each open-source smart contract [37], we eventually collect 119,126 and 278 unique open-source smart contracts in Solidity and Vyper, respectively. These smart contracts were compiled by 155 versions of Solidity compilers and 17 versions of Vyper compilers, e.g., Solidity from V 0.1.1 to V 0.8.0 and Vyper from V 0.1.0b4 to V 0.2.8. Note that we consider a compiler version with optimization and that without optimization as two different versions. Then, we compute the accuracy of `SigRec` for each compiler version. Fig. 15 shows the accuracies in ascending order of Solidity versions, including the number of smart contracts compiled by each version, ranging from 1 to 11,430. Fig. 16 shows the accuracies in ascending order of Vyper versions, which also includes the number of smart contracts compiled by each version, ranging from 1 to 57. We can see that the accuracy of `SigRec` is never lower than 96% for all 155 Solidity compiler versions, and the accuracy of `SigRec` is more than 90% in 12 out of 15 Vyper compiler versions. By investigating the three Vyper compiler versions that make the accuracy of our tool be lower than 90%, we find that the relatively low accuracy is not caused by compiler features. Instead, the small number of Vyper contracts which are compiled by these three versions is the reason. Therefore, the accuracy does not show a downward trend with the evolving of compiler versions.

Answer: *Compiler versions and optimizations bring minimum impact on the accuracy of `SigRec`. Its accuracy is never lower than 96% for all 155 Solidity compiler versions and higher than 90% in 80% of Vyper compiler versions.*

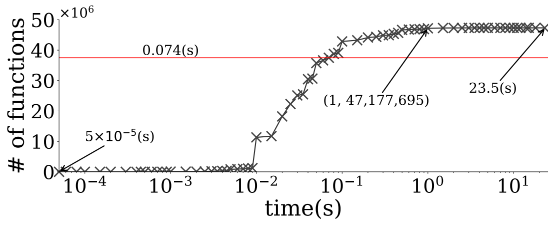

Fig. 17. Time consumption to recover function signatures

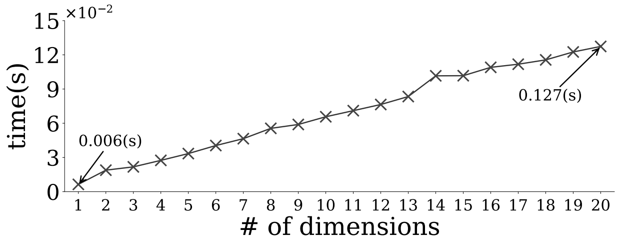

Fig. 18. Time consumption to recover function signatures in different dimensions

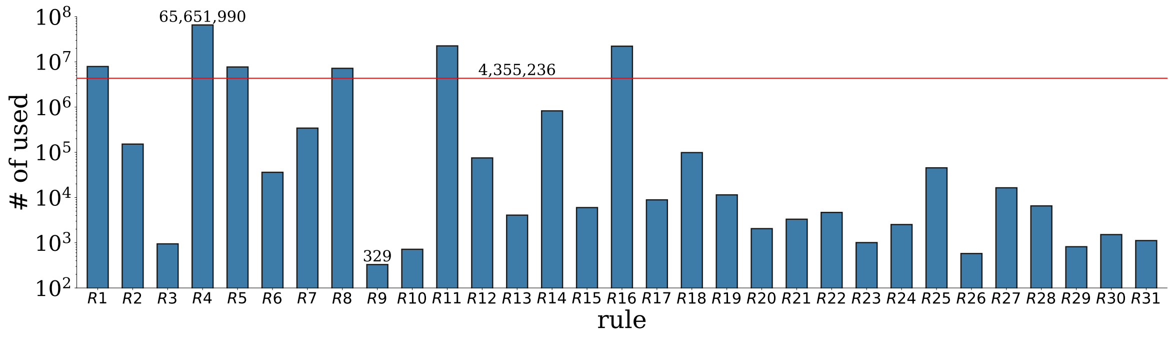

Fig. 19. The number of times each rule is used

5.4 RQ3: How much time is required by `SigRec` to recover function signatures?

We apply `SigRec` to all 47,329,149 public/external functions, and measure the time consumption for recovering each function. This experiment is conducted on a desktop equipped with an Intel Xeon E5-2609 CPU, 16GB main memory and 10TB hard disk. The results are shown in Fig. 17, where each cross (x, y) indicates that `SigRec` uses no more than x seconds to recover each of y function signatures. It shows that the time needed to recover a function signature ranges from 5×10^{-5} seconds to 23.5 seconds, and the average time is 0.074 seconds. For 99.7% $(47,177,695/47,329,149)$ of function signatures, `SigRec` needs no more than 1 second to recover each of them.

We find three reasons for the function signatures which cost long analysis time. First, the analyzed function has many instructions. Second, the analyzed function signature contains any parameter types which will be confirmed after `SigRec` executes all the instructions of the function. Taking uint256 as an example, please recall that we initially assume every basic type as a uint256, and then we change uint256 to another type if the parameter is involved in a special instruction. In other words, `SigRec` recovers the type of parameter as uint256 after it confirms that the parameter is not involved in any special instructions by running all instructions of the analyzed function. On the contrary, `SigRec` can identify some other parameter types quickly because it needs not to run all instructions of the analyzed function. For example, `SigRec` identifies an int$\langle(x+1)\times 8\rangle$ when it encounters a SIGEXTEND instruction.

Third, we find that recovering a higher dimensional array costs more time than a lower dimensional array, because the accessing code for a higher dimensional array contains more bound checks and a larger nested loop for reading array items. To quantitatively study the effect of array dimension, we run `SigRec` to recover an array parameter whose dimension ranges from 1 to 20 and each array item is an uint256. Fig. 18 shows that the time consumption increases linearly along with the increasing of the array dimension. Hence, the array dimension is not a major reason for long analysis time because we find that the array dimension is no larger than 3 in practice.

Answer: *`SigRec` is very efficient. For 99.7% of function signatures, it uses no more than 1 second to recover each of them.*

5.5 RQ4: How frequently is each rule used?

After recovering 47,329,149 public/external functions in all smart contracts, we count how frequently each rule is used. As shown in Fig. 19, all rules have been used. On average, each rule is used for 4,355,236 times. R4 is the most frequently-used rule because the number of basic types in Solidity is more than that of other types. R9 is the least frequently-used rule, because we find that multidimensional static arrays are infrequently used as parameters in public functions.

Answer: *All rules have been used with different frequencies.*

TABLE 1
Results of data set 1

1		Gigahorse	Eveem	OSD	EBD	JEB
2	% same	58.9%	58.1%	18.2%	16.8%	21.3%
3	% fail to process	5.6%	0	0	0	0
4	% no signature	20.8%	20.6%	81.8%	83.2%	78.7%
5	% diff type	12.4%	18.1%	0	0	0
6	% diff num	2.3 %	3.2%	0	0	0

5.6 RQ5: Is `SigRec` superior to existing tools?

We compare `SigRec` with five state-of-the-art decompilers, Gigahorse [38], Eveem [15], OSD [16], EBD [17], and JEB [18] in terms of the accuracy of recovering function signatures.

Datasets. We prepare three datasets for evaluation. Dataset 1 includes all unique closed-source smart contracts. Dataset 2 contains 1,000 synthesized functions. We construct the name of each function with 5 randomly-selected letters, and annotate it as a public function or an external function randomly. Each function takes in x parameters $(1 \le x \le 5)$. For each parameter, we create its name with 5 randomly-selected letters and randomly select a parameter type. Each array parameter has at most three dimensions. Besides, each array parameter has at most five items. The body of each function contains statements to access each parameter, including array items and individual byte of a bytes and a byte32. We construct 100 smart contracts in Solidity, each of which includes 10 synthesized functions, and compile them into bytecode by Solidity 0.5.5 with the probability of 50% to turn on optimization with the default optimization level. Dataset 3 includes all unique open-source contracts obtained from Etherscan.

Results of dataset 1. Table 1 lists the results of all closed-source smart contracts. Since there is no ground-truth for closed-source smart contracts, row 2 lists the ratio of function signatures that the five existing tools produce the same results as `SigRec`. Since tools may abort abnormally when recovering some function signatures, Row 3 presents the

TABLE 2
Results of data set 2

1		Eveem	OSD	EBD	JEB	**SigRec**
2	% correct	18.3%	0%	0%	0%	**98.8%**
3	% fail to process	0	0	0	0	**0**
4	% no signature	1.1%	100%	100%	100%	**0**
5	% type error	65.9%	0	0	0	**0.5%**
6	% parameter num	14.7%	0	0	0	**0.7%**

TABLE 3
Results of data set 3

1		Gigahorse	Eveem	OSD	EBD	JEB	**SigRec**
2	% correct	63.7%	76.2%	44.6%	44.9%	50.1%	**98.7%**
3	% fail to process	3.4%	0	0	0	0	**0**
4	% no signature	25.8%	15.7%	55.4%	55.1%	49.9%	**0**
5	% type error	6.2%	7.3%	0	0	0	**0.5%**
6	% parameter num	0.9%	0.8%	0	0	0	**0.8%**

ratio of such function signatures. Row 4 presents the ratio of function signatures that tools just find their function ids but fail to recover their parameter lists. Row 5 presents the ratio of function signatures that the five tools find the same number of parameters as SigRec, but at least one parameter type recovered by them is different from what is recovered by SigRec. Row 6 presents the ratio of function signatures that the five tools find different parameter numbers from SigRec.

We have two observations from the results. First, the accuracies of OSD, EBD and JEB are low because if their databases do not record a function signature, they cannot recover it. Please note that the databases of OSD, EBD and JEB just cover 31.7% of all function signatures in Ethereum public chain (Supplementary material A). Second, Gigahorse and Eveem outperform OSD, EBD and JEB because they try to infer parameter types if they cannot find function signatures from EFSD [14], [15]. Unfortunately, they still fail to correctly recover a large proportion of function signatures. Since Gigahorse and Eveem are not open-source, we investigate their errors by randomly selecting 50 recovered function signatures with different numbers of parameters and 50 recovered function signatures with different parameter types from SigRec for each of them from data set 1. After checking these 200 function signatures (100 ones are not recovered correctly by Gigahorse, and the other 100 ones are not recovered correctly by Eveem), we observe the following errors: (1) Gigahorse and Eveem report the wrong type. For example, Gigahorse regards the type of a address/uint8/uint256[]/string/bytes parameter as uint256 for some function signatures. Moreover Gigahorse even reports the type of a uint256 parameter as uint2304, which does not exist in smart contracts, in one function signature. (2) Gigahorse mistakenly regards several consecutive parameters as one parameter of a *nonexistent* type. For example, for one function with four parameters whose types are address[], uint256[5], uint256[5] and uint256[2], respectively, Gigahorse outputs only one parameter of uint3328, a nonexistent type. (3) Ghighorse adds more parameters mistakenly. (4) Gigahorse and Eveem miss some parameters. Supplementary material G lists all errors made by Gigahorse and Eveem from the 200 function signatures.

Results of dataset 2. Table 2 lists the results of recovering 1,000 synthesized function signatures by five tools. Row 2 presents the ratio of the function signatures that are correctly recovered. Row 3 and row 4 have the same meaning as the 3rd and 4th rows in Table 1. Row 5 presents the ratio of function signatures that the tools can find the correct parameter numbers but fail to recover the types of at least one parameter. Row 6 presents the ratio of function signatures whose number of parameters cannot be correctly found by the tools. We do not evaluate Gigahorse with the synthesized function signatures because the web service of Gigahorse [38] takes in the addresses of deployed smart contracts rather than the bytecode of smart contracts while deploying 1,000 synthesized functions to Ethereum will cost much money.

The results show that SigRec's accuracy is 98.8%. Manual investigation reveals that the 8 incorrectly recovered function signatures belong to case 5 (§5.2). OSD, EBD and JEB recover 0 function signatures, because none of the synthesized function signatures are recorded in their databases. Eveem correctly recovers 183 synthesized function signatures thanks to its heuristic rules. However, Eveem cannot produce function signatures for 11 functions, outputs incorrect parameter types for 659 function signatures, and outputs incorrect parameter numbers for 147 function signatures. We manually investigate the errors made by Eveem and the results are similar with the investigation of the errors made by Eveem in dataset 1. We present the detailed results in Supplementary material G.

Results of dataset 3. Table 3 lists the results. Row 2 to row 6 have the same meaning with the rows in Table 2. We have several observations. First, SigRec outperforms the other tools by at least 22.5% even the analyzed smart contracts are open-source. Second, Gigahorse is not stable because it aborts abnormally in processing 3.4% of function signatures and it fails to recover some function signatures even they are recorded in EFSD. Third, the accuracies of OSD, EBD and JEB are lower than 51%, even if these functions are implemented in open-source smart contracts. That is, more than 49% function signatures in open-source smart contracts are not recorded in existing function signature databases. Fourth, Eveem outperforms OSD, although they both query EFSD, because Eveem uses its simple rules to infer parameter types if it cannot find function signatures from EFSD. Finally, SigRec outperforms Eveem, because SigRec

TABLE 4
Results of struct and nested array in Solidity

1		Gigahorse	Eveem	OSD	EBD	JEB	**SigRec**
2	% correct	10.1%	10.1%	0%	0%	0%	**61.3%**
3	% fail to process	9.2%	0	0	0	0	**0**
4	% no signature	72.8%	65.5%	100%	100%	100%	**0**
5	% type error	12.0%	20.4%	0	0	0	**26.6%**
6	% parameter num	6.4%	4.0%	0	0	0	**12.4%**

TABLE 5
Results of function signatures in Vyper

1		Gigahorse	Eveem	OSD	EBD	JEB	**SigRec**
2	% correct	68.3%	0%	0%	0%	0%	**97.8%**
3	% fail to process	3.2%	0	0	0	0	**0**
4	% no signature	5.9%	100%	100%	100%	100%	**0**
5	% type error	15.8%	0%	0	0	0	**2.2%**
6	% parameter num	6.8%	0%	0	0	0	**0%**

has a complete set of rules than Eveem and the powerful TASE engine to infer parameter types. We also investigate the errors made by Gigahorse and Eveem by randomly selecting 200 incorrectly recovered function signatures. The results are similar to their errors in processing dataset 1. Supplementary material G contains the detailed results.

Recovery of struct and nested array in Solidity. We compare `SigRec` and existing tools in terms of recovering struct and nested array, which are new parameter types introduced from V 0.4.19 and are supported by the experimental version of Solidity, so-called ABIEncoderV2 before V 0.8.0 [39]. 1,104 function signatures contain struct or nested array in dataset 3, and results are shown in Table 4. We find that the accuracies of existing tools are no higher than 11%, indicating than existing tools do not support these two new parameter types properly. Gigahorse and Eveem have the same accuracy in recovering struct and nested array because these 10.1% of function signatures are recorded in EFSD. In other words, the rules built in these two tools cannot handle struct and nested array. The accuracy of `SigRec` is 61.3%, which is much higher than existing tools. After investigating the function signatures recovered by `SigRec` incorrectly, we find that all of them belong to case 5 (§5.2). Although the accuracy of `SigRec` in recovering struct and nested array is not as high as the accuracy in recovering other types, we believe it is not a big limitation of `SigRec` because the function signatures taking in struct or nested array just account for 0.5% of the total function signatures. We will derive more rules for recovering struct and nested array as our future work.

Recovery of function signatures in Vyper contracts. We compare `SigRec` and existing tools in recovering the function signatures in Vyper contracts, since Vyper is an alternative to Solidity and aims to provide better security than Solidity [40]. 1,076 function signatures appear in all 278 open-source unique Vyper contracts in dataset 3. Results are shown in Table 5. The accuracy of `SigRec` is 97.8%, and we find that all the function signatures which are recovered by `SigRec` incorrectly belong to case 5 (§5.2). Eveem, OSD, EBD and JEB cannot handle Vyper contracts. The accuracy of Gigahorse is 68.3%, and we then investigate the reason for its high accuracy. We find that Gigahorse recovers all Vyper types as uint256 and fortunately, 68.3% function signatures in Vyper contracts taking in uint256 parameters only. Hence, existing tools are inadequate in recovering the function signatures in Vyper contracts.

Importance of function signatures uncovered by existing databases. 289,123, accounting for 75.4% of all unique function signatures are not included in EFSD, the most popular database for function signatures. To investigate whether these uncovered function signatures are important, we count the number of unique bytecode which contains at least one uncovered function signatures. The number of such unique bytecode is 148,268, which accounts for 40.2% of the total unique bytecode, and thus we believe these uncovered function signatures are important. In other words, EFSD misses important function signatures which can be complemented by `SigRec`.

Answer: *`SigRec` is much more accurate than the state-of-the-art tools in recovering function signatures without the need of function signature databases.*

6 Applications of the Recovered Function Signatures

We use three applications to demonstrate the usefulness of `SigRec`, including detecting short address attacks, fuzzing smart contracts, and reverse engineering the bytecode of smart contracts. Due to page limit, we detail the first application and briefly introduce the results of the last two applications here. Interested readers can refer to Supplementary material H, I for more details.

6.1 Detecting Short Address Attacks

We consider the actual arguments sent for function invocation as *invalid* if they are not encoded according to the specification [3]. Although Web3 APIs can generate valid actual arguments, many invalid actual arguments can still be found in practice for various reasons, such as malicious purposes [10], confusion of parameters padding [41], [42], compiler bugs [43], compatibility issues of Web3 [44], [45], changes in the new versions of compiler [46], etc. Invalid actual arguments may incur runtime exceptions, unexpected execution results, and even money stolen [10]. We design ParChecker, the first tool for automatically detecting invalid actual arguments sent to Solidity smart contracts by using the recovered function signatures. We plan to extend ParChecker to support Vyper smart contracts as our future work. Please note that without function signatures, it is difficult to detect them because different parameter types have different padding schemes (§2). For example, to detect short address attacks, we must focus on the functions with address type parameter(s).

ParChecker takes in the call data of a function invocation, and outputs (1) *false* if the actual arguments are invalid, (2) *true* otherwise. ParChecker first gets the function id from the call data, and looks for its function signature from the results of `SigRec`. Then, for each parameter in the function signature, ParChecker locates the actual argument in the call data and checks whether its padding is correct. Specifically, if the parameter is a basic type, ParChecker applies the rules in Table 6 to check the correctness of padding. The rules are derived from the padding scheme of each parameter type described in §2. If the parameter is a static array, ParChecker checks each item of the array according to the rules in Table 6. If the parameter is a dynamic struct, nested array, dynamic array, bytes, or string, ParChecker checks its structure and content. We use a bytes parameter as an example to illustrate the process. ParChecker first locates the *offset* field and the *num* field pointed by the *offset* field in the call data. If either field cannot be found, the structure of the actual argument is invalid. Otherwise, ParChecker reads the *num* field whose value is the length of the bytes before padding, termed by x. After that, ParChecker reads $\lceil x/32 \rceil \times 32$ bytes, termed by v, after the *num* field, and checks whether the lower-order $\lceil x/32 \rceil \times 32 - x$ bits of v are zeros, where "$\lceil\rceil$" denotes rounding up to the next integer. If not, the content of the actual argument is invalid.

We use ParChecker to analyze all transactions in 556,361 blocks starting from the block number 6,625,132. It finds 91,257,261 transactions with non-empty call data and detects 1,024,974 ($1\% = 1,024,974/91,257,261$) transactions with invalid actual arguments. These problematic transactions

TABLE 6
Rules for checking basic types

Type	Rule
uint<M>, $8 \leqslant M \leqslant 256$, M %8==0	The higher-order 256–M bits are zeros.
int<M>, $8 \leqslant M \leqslant 256$, M %8==0	Each bit of the higher-order 256–M bits is equal to the Mth bit.
address	The higher-order 96 bits are zeros.
bool	The higher-order 255 bits are zeros.
bytes<M>, $0<M \leqslant 32$	The lower-order 256–(8×M) bits are zeros.

contain 1,292 unique function ids, invoking 13,556 smart contracts. Among them, we look for the transactions launching short address attacks because they steal tokens [10]. We use such a transaction, which is detected by ParChecker and shown in Fig. 20, to explain how a short address attack can steal tokens, and then present the detection result.

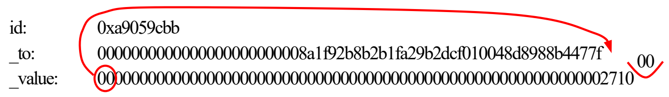

Fig. 20. A short address attack

This transaction invokes the transfer() function of a token smart contract whose function id is 0xa9059cbb to transfer tokens. transfer() has two parameters: *_to* of the type address referring to the receiver, and *_value* of the type uint256 denoting the amount of tokens. The attacker leaves off the trailing zeros of *_to* and then EVM pads *_to* to 32 bytes by the higher-order zeros of *_value*. After that, EVM adds zeros after the lowest byte of *_value* to the length of 32 bytes, and thus *_value* changes from 0x2710 to 0x271000. Attackers can launch such an attack to trick a victim exchange wallet to transfer much more tokens to the address (i.e., *_to*) controlled by them [10]

We discover such attacks from the results of ParChecker by first locating the invocation of transfer(). Then, we check whether the length of actual parameters *len* is shorter than 64 bytes, which is the length of a valid address plus a valid uint256. If so, for the last 32 bytes of the arguments, we check whether its highest $64 - len$ bytes are zeros. If that is the case, a short address attack is detected, because the highest $64 - len$ bytes will be used for complementing the short address. Eventually, we find 73 attacking transactions, which invoke 25 smart contracts. The detailed results are listed in Supplementary material J due to page limit.

Summary: *The function signatures recovered by* `SigRec` *can ease the detection of invalid actual arguments, in particular short address attacks.*

6.2 Boosting the Performance of Existing Smart Contract Fuzzers

It is well-known that knowing parameter types is critical for fuzzing tools to generate high-quality inputs. Many existing fuzzing tools [8], [47], [48], [49] for discovering vulnerabilities in smart contracts assume the availability of function signatures. Unfortunately, such assumption does not always hold because existing tools recover function signatures with low accuracy (§5.6). Without function signatures, existing smart contract fuzzers may have to regard the list of parameters as a byte sequence and generate random byte sequences as input instead of applying specific mutation strategies for different parameter types. `SigRec` can recover function signatures for these fuzzing tools.

To quantitatively evaluate how function signatures recovered by `SigRec` can benefit smart contract fuzzers, we develop a tool, named ContractFuzzer^{-} which is the same with ContractFuzzer [8], a state-of-the-art open-source smart contract fuzzer, except that ContractFuzzer^{-} does not know function signatures and thus it produces random byte sequences as parameters. We compare ContractFuzzer which takes in the function signatures provided by `SigRec` and produces inputs by its default mutation strategies with ContractFuzzer^{-} in analyzing 1,000 randomly selected smart contracts. Results show that with the function signatures recovered by `SigRec`, ContractFuzzer reveals 23% more vulnerabilities and 25% more vulnerable smart contracts than ContractFuzzer^{-}.

Summary: *The function signatures recovered by* `SigRec` *help fuzzers adopt proper fuzzing strategies and find much more vulnerabilities.*

6.3 Improving the Result of Reverse Engineering the Bytecode of Smart Contracts

Erays [50], a reverse engineering tool for smart contracts, takes in the bytecode, and outputs register-based instructions which are more readable than EVM bytecode [50]. Unfortunately, Erays recovers neither function signatures nor variable types, and its output contains lots of code produced by the compiler for accessing parameters, making it difficult to understand the program. We develop a tool named Erays^{+} in 1,456 lines of Python code to improve the results of Erays by leveraging the function signatures recovered by `SigRec`. Specifically, Erays^{+} adds the function signature for each public/external function, replacing meaningless variable names with meaningful parameter names if these variables are copied from parameters (e.g., Erays^{+} replaces a variable x with $arg1$ indicating that x the 1st parameter; Erays^{+} also replaces a variable y with $num(arg1)$ if y is copied from the num field of the 1st parameter), adding the type of the parameter to a variable if the parameter is assigned to the variable, and replacing the bulk of compiler-generated code for accessing parameters with simple assignment statements. Erays^{+} now can leverage the function signatures taking in basics types, static arrays, dynamic arrays, bytes, and strings of Solidity, and we plan to enhance Erays^{+} with the ability to handle structs and nested arrays of Solidity and Vyper types in our future work. Applying Erays^{+} to the bytecode compiled from 53,166 unique open-source contracts, we find that Erays^{+} improves the readability of the outputs of Erays in *all* processed smart contracts. For each of them, the average numbers of added types, added parameter names, added *num* names and removed code lines for accessing parameters are 5.5, 15, 3.4, and 15, respectively.

Summary: *The function signatures provided by* `SigRec` *can be used for improving the readability of the results of reverse engineering tools for smart contracts.*

7 Discussion

This section discusses the limitations of `SigRec` and possible solutions to be explored in future work. First, `SigRec` cannot recover the type of a parameter with the *storage* modifier, because such parameter is a reference to a storage variable with any possible type and the address of the storage variable rather than its content is passed when passing such parameter [2]. The specialty makes the type of such parameter very challenging to be recognized, and we will extend our work to support such parameter in future work. Second, `SigRec` cannot recognize a function parameter if the clues from a function implementation are insufficient. For example, `SigRec` cannot distinguish a bytes from a string, if the smart contract does not access an individual byte of the bytes parameter. We will address this issue by exploiting the huge number of smart contracts, because one function signature may be found in many smart contracts with various function bodies that may provide sufficient clues. That is, one function signature may associate with many function bodies, and thus we may obtain sufficient clues from these function bodies. Third, the accuracy of `SigRec` to recover some parameter types (e.g., nested array) is not as high as the accuracy to recover other types. We will design more rules to infer these types with higher accuracy in future.

Moreover, if a new parameter type or different accessing pattern is introduced, we will determine the rules for recognizing them through the steps described in §3.1. Please recall that there are five steps to derive rules and the first four have been automated so that it will not cost much time to derive new rules. Similarly, malicious smart contracts may use obfuscation to hide their function signatures from being recognized by `SigRec`. A typical obfuscation technique is replacing the instruction sequence for accessing parameters which can be recognized by `SigRec` with a different instruction sequence with the same semantics which cannot be recognized by `SigRec`. We plan to propose general rules to resist obfuscation, where one rule can represent all possible instruction sequences with the same semantics.

8 Related Work

The related studies about recovering function signatures in Smart Contracts have been presented in §1, so this section briefly introduces the related studies about reverse engineering of smart contracts and binaries.

Reverse Engineering Smart Contracts. Being a static analysis framework, Vandal decompiles EVM bytecode into IRs [51]. Porosity decompiles EVM bytecode into readable Solidity syntax contracts [52]. Erays transforms EVM bytecode into human readable expressions [50]. EthIR translates EVM bytecode into a rule-based representation for further analysis [53]. Mythril decompiles EVM bytecode into IRs and performs symbolic execution on IRs [54]. GasReducer disassembles the bytecode into assembly code and detects 24 anti-patterns in the bytecode [55]. [56] captures execution traces of smart contracts to evaluate the number of CFTs covered by traces that are not found by the selected tools. GasChecker disassembles the bytecode into assembly code and performs symbolic execution on the assembly code for further analysis [57]. However, none of them recover function signatures. As a necessary step to discover integer overflow bugs, Osiris applies heuristic rules that are similar to R11 and R13 to infer the types of integer variables [58]. However, Osiris can only recognize unsigned integers and signed integers.

Reverse Engineering Binaries. Reverse engineering techniques highly depend on the runtime architecture of the target programs. EVM bytecode is syntactically similar to Java bytecode. However, Java bytecode retains function signatures [59], and thus does not need to recover function signatures. Next, we briefly introduce the differences between EVM and x86/x64, which demand a new technique to recover function signatures of smart contracts.

First, EVM has 130+ instructions including many blockchain-specific instructions, which have different semantics with x86/x64 instructions [30]. Second, EVM is a stack-based virtual machine without registers [30]. Third, the following concepts are unique to EVM and x86/x64 does not have such design. (1) The call data stores parameters, and only two EVM instructions CALLDATALOAD and CALLDATACOPY can read parameters from the call data [30]. (2) EVM maintains a special memory space named *memory* to store some types of parameters, parameters to internal functions and execution results of smart contracts [30]. Moreover, every smart contract has a permanent storage space named *storage* to record <key, value> pairs [30]. (3) An intra-contract function invocation is compiled into a JUMP instruction, while an inter-contract function invocation is compiled into a CALL/CALLCODE/DELEGATECALL/STATICCALL [30].

Related studies on x86/x64 binaries include recovering parameter types [20], [21], recognizing parameters without recovering parameter types [60], [61], [62], [63], identifying function boundary [64], [65], [66], [67], and inferring variable types [22], [23], [24], [25], [26], [27], [28], [29]. Detailed description is in given Supplementary material K.

9 Conclusion

We propose and develop `SigRec`, a novel approach to automatically recover function signatures in both Solidity smart contracts and Vyper smart contracts without the need of function signature databases. The extensive experimental results show that `SigRec` has very high accuracy across different compilers and various compiler versions. Experiments also show that `SigRec` is efficient and much more accurate than the state-of-the-art tools. Moreover, we demonstrate that the recovered function signatures are very useful in attack detection, fuzzing and reverse engineering of EVM bytecode.

References

[1] Ethereum, "Solidity documentation, — introduction to smart contracts," https://solidity.readthedocs.io/en/latest/introduction-to-smart-contracts.html, 2019.

[2] ——, "Solidity documentation," https://solidity.readthedocs.io/en/latest/index.html, 2019.

[3] ——, "Solidity documentation, — contract abi specification," https://solidity.readthedocs.io/en/latest/abi-spec.html, 2019.

[4] M. di Angelo and G. Salzer, "Wallet contracts on Ethereum," https://arxiv.org/pdf/2001.06909.pdf, 2020.

[5] M. Fröwis, A. Fuchs, and R. Böhme, "Detecting token systems on ethereum," in *FC*, 2019.

[6] M. di Angelo and G. Salzer, "Characterizing types of smart contracts in the Ethereum landscape," in *WTSC*, 2020.

[7] T. Chen, Y. Zhang, Z. Li, X. Luo, T. Wang, R. Cao, X. Xiao, and X. Zhang, "Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum," in *CCS*, 2019.

[8] B. Jiang, Y. Liu, and W. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *ASE*, 2018.

[9] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *CCS*, 2019.

[10] P. Vessenes, "The erc20 short address attack explained," https://vessenes.com/the-erc20-short-address-attack-explained/, 2017.

[11] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, "Safevm: a safety verifier for ethereum smart contracts," in *ISSTA*, 2019.

[12] M. Fröwis and R. Böhme, "In code we trust? measuring the control flow immutability of all smart contracts deployed on ethereum," in *DPM&CBT*, 2017.

[13] 4byte, "Welcome to the ethereum function signature database," https://www.4byte.directory/, 2019.

[14] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis, "Gigahorse: thorough, declarative decompilation of smart contracts," in *ICSE*, 2019.

[15] Eveem, "Eveem," https://eveem.org/, 2019.

[16] ethervm, "Online solidity decompiler," https://ethervm.io/decompile, 2019.

[17] MrLuit, "Evm bytecode decompiler," https://github.com/MrLuit/evm, 2019.

[18] JEB, "Ethereum smart contract decompiler," https://www.pnfsoftware.com/blog/ethereum-smart-contract-decompiler/, 2019.

[19] O. Ganiev, "Ethereum (evm) smart contracts reverse engineering helper utility," https://github.com/beched/abi-decompiler, 2018.

[20] J. Caballero, N. M. Johnson, S. McCamant, and D. Song, "Binary code extraction and interface identification for security applications," in *NDSS*, 2009.

[21] F. de Goër, R. Groz, and L. Mounier, "Lightweight heuristics to retrieve parameter associations from binaries," in *PPREW*, 2015.

[22] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *NDSS*, 2010.

[23] I. Guilfanov, "Simple type system for program reengineering," in *WCRE*, 2001.

[24] J. Lee, T. Avgerinos, and D. Brumley, "Tie: Principled reverse engineering of types in binary programs," in *NDSS*, 2011.

[25] K. Troshina, Y. Derevenets, and A. Chernov, "Reconstruction of composite types for decompilation," in *SCAM*, 2010.

[26] M. Noonan, A. Loginov, and D. Cok, "Polymorphic type inference for machine code," in *PLDI*, 2016.

[27] O. Katz, R. El-Yaniv, and E. Yahav, "Estimating types in binaries using predictive modeling," *POPL*, 2016.

[28] Z. Xu, C. Wen, and S. Qin, "Learning types for binaries," in *ICFEM*, 2017.

[29] A. Slowinska, T. Stancescu, and H. Bos, "Howard: A dynamic excavator for reverse engineering data structures." in *NDSS*, 2011.

[30] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger byzantium version bdec180 – 2019-05-01," https://ethereum.github.io/yellowpaper/paper.pdf, 2019.

[31] A. Beregszaszi and P. Bylica, "Bitwise shifting instructions in evm," https://github.com/ethereum/EIPs/blob/master/EIPS/eip-145.md, 2017.

[32] Ethereum, "Solidity documentation, — types," https://solidity.readthedocs.io/en/latest/types.html, 2019.

[33] ——, "Vyper documentation," https://vyper.readthedocs.io/en/latest/, 2021.

[34] ——, "Go ethereum – official golang implementation of the ethereum protocol," https://github.com/ethereum/go-ethereum, 2019.

[35] T. Chen, Z. Li, Y. Zhu, J. Chen, X. Luo, J. C.-S. Lui, X. Lin, and X. Zhang, "Understanding ethereum via graph analysis," in *INFOCOM*, 2018.

[36] T. Chen, Z. Li, Y. Zhang, X. Luo, A. Chen, K. Yang, B. Hu, T. Zhu, S. Deng, T. Hu *et al.*, "Dataether: Data exploration framework for ethereum," in *ICDCS*, 2019.

[37] Ethereum, "Etherscan — ethereum (eth) blockchain explorer," https://etherscan.io/, 2019.

[38] DEDAUB, "Contract library," https://contract-library.com/, 2019.

[39] Ethereum, "Solidity documentation, — changelog," https://github.com/ethereum/solidity/blob/develop/Changelog.md, 2021.

[40] M. Kaleem, A. Mavridou, and A. Laszka, "Vyper: A security comparison with solidity based on common vulnerabilities," in *BRAINS*, 2020.

[41] Kristaps, "Solidity, problem with bytes decoding," https://ethereum.stackexchange.com/questions/69713/solidity-problem-with-bytes-decoding, 2019.

[42] E. Luis, "encode data input of the raw transaction to update an contract function?" https://ethereum.stackexchange.com/questions/25144/encode-data-input-of-the-raw-transaction-to-update-an-contract-function, 2017.

[43] MeMyself, "Solidity "call" function with array as input," https://ethereum.stackexchange.com/questions/16144/solidity-call-function-with-array-as-input, 2017.

[44] filips123, "Compatibility with newer web3 versions," https://github.com/trufflesuite/truffle/issues/1690, 2019.

[45] taducquang, "Can't install web3 version 1.2 on windows 10," https://github.com/ethereum/web3.js/issues/2971, 2019.

[46] Ethereum, "Solidity v0.5.0 breaking changes," https://solidity.readthedocs.io/en/v0.5.0/050-breaking-changes.html, 2018.

[47] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: An efficient adaptive fuzzer for solidity smart contracts," in *ICSE*, 2020.

[48] V. Wüstholz and M. Christakis, "Targeted greybox fuzzing with static lookahead analysis," in *Proc. ICSE*, 2020.

[49] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to Fuzz from Symbolic Execution with Application to Smart Contracts," in *Proc. CCS*, 2019.

[50] Y. Zhou, D. Kumar, S. Bakshi, J. Mason, A. Miller, and M. Bailey, "Erays: Reverse engineering ethereum's opaque smart contracts," in *USENIX Security Symposium*, 2018.

[51] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," https://arxiv.org/pdf/1809.03981.pdf, 2018.

[52] M. Suiche, "Porosity: A decompiler for blockchain-based smart contracts bytecode," in *DEFCON 25*, 2017.

[53] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, "Ethir: A framework for high-level analysis of ethereum bytecode," in *ATVA*, 2018.

[54] ConsenSys, "Mythril," https://github.com/ConsenSys/mythril, 2019.

[55] T. Chen, Z. Li, H. Zhou, J. Chen, X. Luo, X. Li, and X. Zhang, "Towards saving money in using smart contracts," in *ICSE-NIER*, 2018.

[56] T. Chen, Z. Li, Y. Zhang, X. Luo, T. Wang, T. Hu, X. Xiao, D. Wang, J. Huang, and X. Zhang, "A large-scale empirical study on control flow identification of smart contracts," in *ESEM*, 2019.

[57] T. Chen, Y. Feng, Z. Li, H. Zhou, X. Luo, X. Li, X. Xiao, J. Chen, and X. Zhang, "Gaschecker: Scalable analysis for discovering gas-inefficient smart contracts," *IEEE Transactions on Emerging Topics in Computing*, 2020.

[58] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in Ethereum smart contracts," in *ACSAC*, 2018.

[59] J. Hamilton and S. Danicic, "An evaluation of current java bytecode decompilers," in *SCAM*, 2009.

[60] K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua, "Scalable variable and data type detection in a binary rewriter," in *PLDI*, 2013.

[61] K. Anand, M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua, "A compiler-level intermediate representation based binary analysis and rewriting system," in *EuroSys*, 2013.

[62] J. Zhang, R. Zhao, and J. Pang, "Parameter and return-value analysis of binary executables," in *COMPSAC*, 2007.

[63] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *CC*, 2004.

[64] R. Qiao and R. Sekar, "Function interface analysis: A principled approach for function recognition in cots binaries," in *DSN*, 2017.

[65] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "Byteweight: Learning to recognize functions in binary code," in *USENIX Security Symposium*, 2014.

[66] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *USENIX Security Symposium*, 2015.

[67] S. Wang, P. Wang, and D. Wu, "Semantics-aware machine learning for function recognition in binary code," in *ICSME*, 2017.