Precise Static Modeling of Ethereum "Memory"

SIFIS LAGOUVARDOS, University of Athens, Greece
NEVILLE GRECH, University of Athens, Greece
ILIAS TSATIRIS, University of Athens, Greece
YANNIS SMARAGDAKIS, University of Athens, Greece

Static analysis of smart contracts as-deployed on the Ethereum blockchain has received much recent attention. However, high-precision analyses currently face significant challenges when dealing with the Ethereum VM (EVM) execution model. A major such challenge is the modeling of low-level, transient "memory" (as opposed to persistent, on-blockchain "storage") that smart contracts employ. Statically understanding the usage patterns of memory is non-trivial, due to the dynamic allocation nature of in-memory buffers. We offer an analysis that models EVM memory, recovering high-level concepts (e.g., arrays, buffers, call arguments) via deep modeling of the flow of values. Our analysis opens the door to Ethereum static analyses with drastically increased precision. One such analysis detects the extraction of ERC20 tokens by unauthorized users. For another practical vulnerability (redundant calls, possibly used as an attack vector), our memory modeling yields analysis precision of 89%, compared to 16% for a state-of-the-art tool without precise memory modeling. Additionally, precise memory modeling enables the static computation of a contract's gas cost. This gas-cost analysis has recently been instrumental in the evaluation of the impact of the EIP-1884 repricing (in terms of gas costs) of EVM operations, leading to a reward and significant publicity from the Ethereum Foundation.

CCS Concepts: • **Software and its engineering → Formal software verification**; • **Theory of computation → Program analysis**.

Additional Key Words and Phrases: ethereum, EVM, static analysis

ACM Reference Format:
Sifis Lagouvardos, Neville Grech, Ilias Tsatiris, and Yannis Smaragdakis. 2020. Precise Static Modeling of Ethereum "Memory". *Proc. ACM Program. Lang.* 4, OOPSLA, Article 190 (November 2020), 26 pages. https://doi.org/10.1145/3428258

1 INTRODUCTION

The Ethereum blockchain has enabled the management of digital assets via unsupervised autonomous agents called *smart contracts*. Smart contracts are among the computer programs with the most dire needs of high correctness assurance, due to their managing of high-value assets, as well as their public and immutable nature. Therefore, static analysis for Ethereum smart contracts has captured the attention of the research community in recent years [Albert et al. 2018; Feist et al. 2019; Grech et al. 2018; Mueller 2018; Tsankov et al. 2018]. The first generation of Ethereum

Authors' addresses: Sifis Lagouvardos, Department of Informatics and Telecommunications, University of Athens, Athens, Greece, sifis.lag@di.uoa.gr; Neville Grech, Department of Informatics and Telecommunications, University of Athens, Athens, Greece, me@nevillegrech.com; Ilias Tsatiris, Department of Informatics and Telecommunications, University of Athens, Athens, Greece, i.tsatiris@di.uoa.gr; Yannis Smaragdakis, Department of Informatics and Telecommunications, University of Athens, Athens, Greece, yannis@smaragd.org.

2475-1421/2020/11-ART190
https://doi.org/10.1145/3428258

static analyses have already exhibited significant successes.[1] For instance, Securify [Tsankov et al. 2018] has been the basis for a leading company in the Ethereum security space, with many tens of commercial audits performed.

Most static analysis tools for Ethereum operate at the binary level of contracts, as-deployed on the blockchain. This ensures that the analysis operates on all contracts in existence, regardless of whether source code is available. (Source code is not always present but not negligible either: it is available for under 25% of deployed contracts, yet for more than half of the high-value contracts.) Furthermore, operating on low-level binaries offers source-language and language-version independence, completeness in the presence of inline assembly (which is common), and, perhaps most importantly, uniform treatment of complex language features: as analysis authors often argue, innocuous-looking source-level expressions can incur looping behavior or implicit overflow [Grech et al. 2018].

Operating at the binary level necessitates contract decompilation [eth [n. d.]; Brent et al. 2018; Grech et al. 2019; Kolinko 2018] in order to recover high-level information from the very-low-level Ethereum VM (EVM) bytecode format. This decompilation effort is non-trivial: the EVM is a stack machine with no structured information (no types or functions). Control-flow (i.e., calls, returns, conditionals) is implemented as forward jumps over run-time values obtained from the stack, hence even producing a control-flow graph requires a deep static analysis [Grech et al. 2018]. Despite these challenges, sophisticated decompilers mostly succeed in recovering high-level control flow and have been the basis of the most successful static analyses.

Despite the relative success of Ethereum decompilers, some low-level aspects of the deployed contract remain unaddressed. The most major such aspect is the precise modeling of transient EVM *memory*. "Memory" in the context of the EVM (and of this paper) refers to a data store that is transient and transaction-private, used by the runtime primarily to store values whose size is statically unknown—e.g., arrays, strings, or encoded buffers. (Memory is to be contrasted with *storage*: the on-blockchain persistent value store of an Ethereum smart contract.)

Memory is crucial in the execution of smart contracts. It is used for many cryptographic hash operations (`SHA3` is a native instruction, operating over arbitrary-length buffers), for the arguments of external calls (i.e., calls to other contracts), for logging operations, and for returns from a public function. Crucially, any complex mapping data structure (i.e., the most common Ethereum data structures) stores and retrieves information (from *storage*) after hashing keys in *memory* buffers.

Modeling EVM memory is unlike other memory management settings. There is no speed premium for locality, yet the compiler attempts to keep memory tightly packed because the contract's execution cost (in terms of Ethereum *gas* consumed for memory-related operations) depends on the highest initialized memory address. For the vast majority of smart contracts, written in the Solidity language, the compiler uses a simple strategy: every allocated buffer is written after the end of the last one, updating a "free-memory" pointer. This memory management policy is a low-level aspect, introduced entirely during the compilation process. At the source-code level, memory is implicit: the values that end up stored in memory merely have dynamic-length array or string types.
The goal of a precise modeling of EVM memory is to recover such source-level information (expressed in terms of arrays or strings) from the low-level address manipulation code that the compiler produces.

[1] Terminology-wise, we make a distinction between static analysis tools, i.e., tools that aim for complete modeling of all possible executions, vs. symbolic-execution tools. Symbolic execution also performs static reasoning but follows only the execution paths that it can model precisely, with specific values. The context of most of our presentation is static analysis. We compare to symbolic execution tools (which exhibit high precision but low completeness) in our Related Work discussion—Section 7.

Generally, our work makes the following contributions:

- We introduce the problem of statically modeling EVM memory and posit that it is a key component of high-precision, next-generation static analyses of Ethereum smart contracts.
- We propose an approach to memory modeling, in conjunction with a regular flow analysis, recognizing symbolic relationships between non-constant memory addresses and other data (e.g., array or string lengths).
- We introduce three practical, novel analyses that vastly benefit from precise memory modeling: a tainted ERC20 token `transfer` analysis, detecting the extraction of ERC20 tokens by unauthorized malicious parties, a redundant calls analysis (which can be an indication of insidious vulnerabilities, as in the DAO Stack bounty attack in 2019 [Levi 2019]) and a precise static gas cost estimation, specifically evaluated over the Ethereum operations gas repricing introduced in Ethereum Enhancement Proposal (EIP) 1884.
- The latter analysis was instrumental in the discussions of the impact of EIP-1884 leading to a bounty and publicity by the Ethereum Foundation. It remains the only static analysis precise enough to pinpoint contracts crucially affected by the repricing of gas operations. A static analysis is invaluable in answering this question because typically no past transaction has exercised the functionality (i.e., low-gas calls to the fallback function) affected by the repricing.

Illustration. Consider a simple code fragment declaring a field and a method over it:[2]

```
1 mapping(string => string) mTokens; ...
2 function getToken(string pDocumentHash) view public returns(string)
3 { return mTokens[pDocumentHash]; }
```

This seemingly simple method gets translated into well-over-100 EVM bytecode instructions, or several tens of lines of decompiled three-address code, under the popular decompilers [eth [n. d.]; Dedaub 2019; Kolinko 2018]. Figure 1 lists the version produced by EtherVM, which shows all the elements we refer to, such as multiple loops (back-jumps highlighted), memory, and cryptographic hash operations (`keccak256`, a.k.a. `SHA3`).

The code operates over memory, since strings are of unknown size. Specifically, the code first allocates a memory array (updating the free-memory pointer at $0x40$) and fills it in a loop that iterates over the contents of `pDocumentHash`. Subsequently, another in-memory buffer is allocated and filled (via a loop) with the id of mapping `mTokens` followed by the contents of the same string as before. A cryptographic hash (`keccak256`) over the latter buffer is run to compute the storage address that holds the mapping's value for the given key. Then the result is again copied (via another loop) in a freshly-allocated in-memory buffer. Finally, another buffer gets allocated in order to do the ABI encoding of the value, i.e., to encode it in the form necessary for returning to the external caller of the contract's method. This memory buffer also gets filled via a loop.

Nearly all memory load and store operations in the above code are to non-constant addresses computed by looping—we discuss in Section 3 how past approaches fail to model them. Modeling such behavior requires both a precise analysis of value flow *and* a modeling of high-level concepts, such as "this address holds an array", "this address holds the length of that array", and even much more complex, such as "this address is unknown but corresponds to the original free-memory pointer plus the length of the input array". Our approach succeeds in capturing such abstract relationships. This opens the door for several high-value applications, such as precise gas consumption analysis and precise analysis for duplicate calls. We postpone their illustration until Section 5.

[2]Code excerpted from deployed contract, https://etherscan.io/address/0x212ec09a66fdd550e62794c35b9b347dbd6a5a90 .

```
function getToken(var arg0) returns (var r0) {
  var var0 = 0x053b;  var0 = func_06C6();  var var1 = 0x02;
  var temp0 = arg0;  var var2 = temp0;
  var var3 = memory[0x40:0x60];  var var4 = var3;
  var var5 = var2 + 0x20;  var var6 = memory[var2:var2 + 0x20];
  var var7 = var6;  var var8 = var4;  var var9 = var5;
  if (var7 < 0x20) {
  label_0573:
    var temp1 = 0x0100 ** (0x20 - var7) - 0x01;  var temp2 = var8;
    memory[temp2:temp2 + 0x20] = (memory[var9:var9 + 0x20] & ~temp1) | (memory[temp2:temp2 + 0x20] & temp1);
    var temp3 = var6 + var4;
    memory[temp3:temp3 + 0x20] = var1;
    var temp4 = memory[0x40:0x60];
    var temp5 = keccak256(memory[temp4:temp4+(temp3+0x20)-temp4]);
    var temp6 = storage[temp5];
    var temp7 = (!(temp6 & 0x01) * 0x0100 - 0x01 & temp6) / 0x02;
    var temp8 = memory[0x40:0x60];
    memory[0x40:0x60] = temp8 + (temp7+0x1f) / 0x20 * 0x20 + 0x20;
    var1 = temp8;  var2 = temp5;  var3 = temp7;
    memory[var1:var1 + 0x20] = var3;
    var4 = var1 + 0x20;  var5 = var2;
    var temp9 = storage[var5];
    var6 = (!(temp9 & 0x01) * 0x0100 - 0x01 & temp9) / 0x02;
    if (!var6) {
    label_063A:
      return var1;
    } else if (0x1f < var6) {
      var temp10=var4; var temp11 = temp10 + var6; var4=temp11;
      memory[0x00:0x20] = var5;
      var temp12 = keccak256(memory[0x00:0x20]);
      memory[temp10:temp10 + 0x20] = storage[temp12];
      var5 = temp12 + 0x01;  var6 = temp10 + 0x20;
      if (var4 <= var6) { goto label_0631; }
    label_061D:
      var temp13 = var5;  var temp14 = var6;
      memory[temp14:temp14 + 0x20] = storage[temp13];
      var5 = temp13 + 0x01;  var6 = temp14 + 0x20;
      if (var4 > var6) { goto label_061D; }
    label_0631:
      var temp15 = var4; var temp16 = temp15+(var6 - temp15&0x1f);
      var6 = temp15;  var4 = temp16;
      goto label_063A;
    } else {
      var temp17 = var4;
      memory[temp17:temp17+0x20] = storage[var5]/0x0100 * 0x0100;
      var4 = temp17 + 0x20;  var6 = var6;
      goto label_063A;
    }
  } else {
  label_0559:
    var temp18 = var9;  var temp19 = var8;
    memory[temp19:temp19 + 0x20] = memory[temp18:temp18 + 0x20];
    var8 = temp19 + 0x20; var9 = temp18 + 0x20; var7 = var7-0x20;
    if (var7 < 0x20) { goto label_0573; }
    else { goto label_0559; }
  }
}
```

Fig. 1. Function `getToken` decompiled. **The reader is not expected to follow this code closely, beyond appreciating the low-level complexity hidden behind seemingly-simple high-level statements.**

2 BACKGROUND

We next provide brief background on the Ethereum Virtual Machine (EVM) execution and its cost model.

Transaction execution environment. Ethereum is a *programmable blockchain*: it is a decentralized register of transactions that also serves as the execution environment for smart contracts, which are programs associated with an account address. Transaction invocations in Ethereum contain not only the basic parameters for cryptocurrency transfer between addresses but also executions of smart contracts over user-supplied data. For illustration, an Ethereum user (a *sender*) can submit

a transaction proposal to the Ethereum network containing: (i) `sender` and `receiver` fields (ii) a message (iii) value and (iv) a gas budget, e.g., (simplified):

i	`sender: 0xBEEFBABE receiver: 0xC0CAC01A`
ii	`deliverCans(amount: 10, to: 0xCAFED00D)`
iii	2 ETH ($\sim$ \$400)
iv	10000 gas units @ $2 * 10^{-7}$ETH each

This transaction proposal is signed using the sender's private key (i.e., one that corresponds to the private key `0xBEEFBABE`) and placed in a distributed buffer called the *mempool*. Miners execute transactions from this mempool *if* the supplied gas price (part of (iv)) is acceptably high. The executed transaction then appears as part of the next *block*, an ordered sequence of transactions. Each block also contains a hash of the previous block, all the way to the genesis block—a key mechanism to ensure the consistency of the blockchain.

In order to execute these messages, (ii), the address (`0xC0CAC01A`) should be holding a smart contract. Smart contracts are executed using the EVM—a distributed virtual machine, described in a precise specification. The EVM calculates the execution cost (gas) of this message instruction by instruction. This cost is paid for by the gas budget provided by the sender (iv).

Memory and Storage. The EVM memory model distinguishes between *storage* and *memory*. Storage is a persistent data store, on the blockchain, whereas memory is transient, zero-set when any transaction is started. The EVM word length is 256-bits, and memory is a word-addressed array of bytes, whereas storage is a word-addressed array of words.

The economics, in terms of gas prices, of storage and memory are different. Storing a single word in storage costs up to 20000 units of gas, whereas in memory this would cost 3 units of gas per byte. The billing mechanism is also different: a fee is paid for the total size of the current memory set, which is proportional to the highest address of any non-zero word. No similar fee exists for storage. This fee therefore incentivizes the compiled contract to pack as much data as possible into the lower memory addresses. The compiler does so by baking in logic to track where memory is free using a *free-memory pointer*, which is also stored in memory. However, no incentive exists to compact storage space. Hence, it is fairly common to make use of cryptographic hash functions (e.g., `SHA3`, which returns average values in the order of 2^{128}) to compute pointer addresses in storage space to write to.

Memory is used as a temporary store for all sorts of data—function parameters, transient data structures, scratch space, etc. Modeling memory is therefore necessary for many client analyses. Memory modeling is also necessary to deeply analyze the semantics of various instructions. For instance, memory is used as a temporary data buffer for the input of `SHA3` instructions. External `CALL` instructions use memory to pass a message (which includes the function signature and arguments) and also for returning data from these (ditto for the `RETURN` opcode). Other instructions for logging messages on the platform, e.g., `LOG1` use memory.

`CALL` instructions, which actually start a new *internal transaction* on the EVM, encode parameters and other data using a standard Application Binary Interface (ABI) encoding. To recover the parameters of a `CALL` and `RETURN` instruction, memory modeling needs to be complemented with a general model of the Ethereum ABI. This ABI specifies how variables of different sizes (including variable length arrays) are encoded.

3 STATE-OF-THE-ART

First-generation Ethereum static analyses could get away without a precise modeling of memory. We illustrate with the contrasting approaches of Securify [Tsankov et al. 2018],[3] EthIR [Albert et al. 2018], and MadMax [Grech et al. 2018].

Securify implements a standard modeling of memory that favors *completeness* while sacrificing precision. Specifically memory loads are modeled as [Tsankov et al. 2018, Fig.8]:[4]

```
MayDepOn(y,t) :- l:[y := MLOAD(o)], isConst(o), MemTag(l,o,t).
MayDepOn(y,t) :- l:[y := MLOAD(o)], !isConst(o), MemTag(l,_,t).
MayDepOn(y,t) :- l:[y := MLOAD(o)], MemTag(l,⊤,t).
```

That is, a load instruction from a constant offset (i.e., address) o results in a dependency of the variable reading the value on the registered tag of the memory. Imprecision is introduced, however, when the address is not constant, in which case any memory tag at that instruction label, l, can be returned; as well as when there is a tag for a $\top$ ("any") address—introduced for stores to unknown addresses:

```
MemTag(l,o,t) :- l:[MSTORE(x,o)], isConst(o), MayDepOn(x,t).
MemTag(l,⊤,t) :- l:[MSTORE(x,o)], !isConst(o), MayDepOn(x,t).
```

That is, a store to a non-constant address at instruction label l attaches a tag t to any ($\top$) address.

The precision of this modeling depends crucially on how many of the instructions load or store from/to known, constant addresses. Initially, during the decompilation phase, Securify employs constant folding and propagation in order to recover constant memory offsets. This approach relies on being able to compute concrete values for the free-memory pointer and should be able to produce precise offsets in programs that do not use variable-length types. In programs that make use of variable-length types, such as string or bytes in Solidity, the free-memory pointer will be incremented by a non-statically computable value causing all operations following one that uses a variable-length type to have unknown indexes. The above combined with the handling of `MLOAD` and `MSTORE` operations results in an analysis that favors completeness while sacrificing precision.

Despite this imprecision, Securify reports [Tsankov et al. 2018, Figure 14] that the offsets of 80% of the `MSTORE` instructions are statically resolved. To put this number in perspective, however, we measure experimentally that 74% of `MSTORE` instructions have a constant index, as computed via simple constant folding and local propagation. As one can guess, these are the easy, shallow cases of memory use. The interesting cases (e.g., requiring accurate modeling of the signature and arguments of an external `CALL` instruction), which arise in all client analyses we will consider in our evaluation, are not modeled at all via this strategy—they require modeling abstract expressions relative to the free-memory pointer.

Conversely, MadMax [Grech et al. 2018] takes memory into account with more precision but incompletely, only in well-defined, intra-procedural settings. There is no computation of unknown memory contents, only of the ways that a memory value can be related to a specific storage data structure.[5]

[3] Our discussion pertains to the original Securify tool. The recent Securify2 [ChainSecurity 2020] is a source-level tool, which cannot apply to contracts in compiled form. Therefore no memory modeling discussion is applicable: the tool is relieved of the complexity that our work addresses, at the expense of only applying to a tiny fraction of contracts in the wild. Brent et al. [2020] measure the applicability of Securify2 to under 3% of currently deployed contracts.

[4] We use a Datalog formulation for the analysis specification, much like the closest comparable prior art: Securify and MadMax. Datalog rules are implications where any matching values of variables in the rule body (to the right of the ":- " sign) produce an inference of an instance of the relation on the rule head (to the left of ":- ").

[5] The rule is from the public MadMax repository, in https://github.com/nevillegrech/MadMax/blob/master/tools/bulk_analyser/spec.dl .

```
KeyToOffset(keyVar, storeOffsetVar, keySize) :-
  shaStmt:[storeOffsetVar := SHA3(shaStart, keySizeVar)],
  LocalAlias(mstoreStmt, shaStart2, shaStmt, shaStart),
  mstoreStmt: [MSTORE(shaStart2, keyVar)],
  value(keySizeVar, keySize),
  FlowsFrom(index, storeOffsetVar),
  StorageIndex(index).
```

That is, store instructions are only modeled when their target address is locally used to compute a SHA3 cryptographic hash. (LocalAlias is only computed for variables in the same function, hence it is relatively precise but incomplete.) In this case, the data written are considered to be a mapping data structure's key. This approach may be precise enough for MadMax's purposes, but other high-value analyses cannot leverage it: no computation is made of either the size of data or their precise identity.

Finally, the EthIR [Albert et al. 2018] framework follows a similar approach to MadMax, favoring precision but suffering incompleteness. It annotates MSTORE and MLOAD instructions with the memory address they operate on when this address can be computed statically and attempts to transform these addresses into variables. The authors of EthIR acknowledge incompleteness when it comes to arrays, noting that their approach will fail *"when we have an array access inside a loop with a variable index"*.

Our approach is both more precise and more complete than such earlier treatment, enabling client analyses not previously possible.

4 PRECISE MEMORY MODELING

We next describe our approach to smart contract memory modeling. Producing an analysis that infers the necessary high-level information is as important as *identifying* what is such information. The informal goal of the analysis is to eliminate MSTORE and MLOAD instructions (including their dependent instructions, all the way to loops that include them) and replace them with high-level operations on arrays, ABI-encoded buffers, or strings. The key tenets of the analysis are to (a) leverage information from other flow, alias, dominance, etc. analyses for precision; (b) infer full information on sizes and index patterns when possible, but fall back gracefully to inferring still-insightful partial information in the case of too-complex iteration.

4.1 Overview

The analysis inference starts from detecting the root memory allocation pattern. For Solidity—the dominant Ethereum smart contract implementation language—the pattern is simply a bump-pointer allocation. (Notably, there is never deallocation under the Solidity compiler. This is hardly surprising, since transactions are short lived and gas-limited.) A *free-memory pointer* is maintained at a constant memory address ($0x40$, or decimal 64) and incremented at every allocation to point to the address of the first un-allocated word. (Notably, the address pointed by the free-memory pointer is often used as a scratchpad, without adjusting the pointer. Hence, "free-memory pointer" is somewhat of a misnomer.) Although Solidity dominates the Ethereum development space,[6] our core analysis is agnostic to the specific pattern used to detect such an upper bound/free-memory pointer: any such syntactic inference is accepted as input.

[6] Etherscan.io currently contains the largest collection of smart contracts with source code. It supports the two most popular languages: Solidity and Vyper. As of mid-May 2020, there were a mere 74 Vyper contracts on Etherscan, compared to some-80,000 Solidity contracts. Though unlikely, it is conceivable that contracts with no released source code (which are still 3-to-4× more than those *with* source code) are written in other languages, but, given the ever-increasing Solidity dominance, it is unlikely that such languages will use memory conventions incompatible with Solidity libraries.

V is a set of program variables	S is a set of statement identifiers	M is a set of symbolic values
C is a set of constants, $C \subseteq \mathbb{N}_{256}$	$\mathbb{N}_{256}$ is the set of 256-bit unsigned integers	**FreePtr** is the free-memory pointer

Concrete instructions	
(s: S) :[$r : V$:= ADD/SUB/MUL(a: $V \cup C$, b: $V \cup C$)]	s is instruction ADD/SUB/MUL with arguments a, b, and result r
(s: S) :[MSTORE($addr : V$, $from : V$)]	s is memory store of value *from* to address *addr*
(s: S) :[$to : V$:= MLOAD($addr : V$)]	s is memory load to variable *to* from address *addr*
(s: S) :[$to : V$:= CALLDATALOAD($arg : V$)]	s reads one word from call-arguments area at offset *arg* into *to*
(s: S) :[CALLDATACOPY($toAddr : V$, $arg : V$, $len : V$)]	s copies to memory at *toAddr len* words from offset *arg* of call-arguments area
Generic instructions / Syntactic patterns	
StatementUsesMemory($s : S$, $start : V$, $len : V$)	denotes instructions (e.g., SHA3 or LOG) operating over memory
Interfacing with other analysis modules	
Variable_Value($v : V$, $c : C$)	**Constant-propagation/folding analysis**: v holds const. value c
Flows(*from* : V, $to : V$)	**Flow analysis**: the value of *from* is used in the computation of *to*
Alias($x : V$, $y : V$)	**Alias analysis**: the two variables hold identical values
MatchingMSTORE($ms : S$, $s : S$)	**Program intervals analysis**: no memory-accessing statements between MSTORE *ms* and memory-consuming statement *s*
UnchangedFreePointer($s1 : S$, $s2 : S$)	no memory allocation takes place between two statements
Computed (intermediate or output) relations	
Variable_Value$^+$($v : V$, $x : M \cup C$)	extension of constant-folding relation to include symbolic values
FreePointerBasedValue($x : M$, $s : S$, $c : C$)	x is symbolic value equal to free-memory pointer (read at s) + c
FreePointerDiff($x1 : M$, $x2 : M$, $diff : C$)	two free-pointer-based symbolic values differ by *diff*
IsArrayVar($v : V$)	v is inferred to be the address of an array
Array_ElementSize($a : V$, $c : C$)	array at address a contains elements of size c
ArrayIndexAccess($a : V$, $iL : V$, $iH : V$)	array at address a is accessed via low-level address variable *iL*, mapping to (high-level) element index *iH*
ArrayLoadAtIndex($mld : S$, $a : V$, $iH : V$, $to : V$)	MLOAD statement *mld* reads into *to* from a at element index *iH*
ArrayStoreAtIndex($mst : S$, $a : V$, $iH : V$, $from : V$)	MSTORE statement *mst* writes *from* into a at element index *iH*
ArrayLoad($mld : S$, $a : V$, $to : V$)	MLOAD statement *mld* reads into *to* from a at undetermined index
ArrayStore($mst : S$, $a : V$, $from : V$)	MSTORE statement *mst* writes *from* into a at undetermined index
MemoryStatement_ActualArg($mst : S$, $i : C$, $arg: V$)	memory consuming statement *mst* operates on *arg* at index i

Fig. 2. Domains, input, connecting, and computed relations.

The analysis then tracks, through the contract code, symbolic values based on the free-memory pointer, offset by a constant. Interesting concepts arise during this analysis, such as aliased variables, program intervals in which the free-memory pointer remains unchanged, and more. The patterns of inferring high-level constructs (arrays or strings) are based on such symbolic values. The analysis uses these values while recognizing array allocations, uses of arrays in language constructs that consume memory (SHA3, CALL/DELEGATECALL, LOG, RETURN, REVERT) and more. The specific memory encodings vary per-construct. One key pattern is the use of the EVM Application Binary Interface (ABI), which dictates the encoding of parameters passed between contracts.

The power of the analysis is due to corroborating evidence from multiple sources, in order to infer with high confidence that allocated data should be treated as a unit, i.e., as a single higher-level array, buffer, or string.

4.2 Preliminaries and Schema

We present a model of the analysis as Datalog inference rules. Although the full implementation is much more complex, the goal of the model is to capture the essential complexity of the approach, for representative language features. We elide aspects that have mostly engineering complexity, or that are well-represented by other concepts.

The analysis operates on a low-level program representation. The inputs used in our analysis model are shown in Figure 2, together with relations interfacing with other analysis modules, and computed relations. A few points are worth elaborating:

- Arithmetic operations implicitly assume the application of algebraic equalities and constant folding for normalization of the program text—e.g., we write *stmt:* [y := ADD(x, *32*)], hiding the

intermediate variable whose value is inferred to be *32*, as well as that the addition in the program code could be either *x + 32* or *32 + x*.
- Although memory-consuming instructions (e.g., `SHA3`) lead to specialized inferences in the implementation, in the analysis model they do not need to be distinguished and are unified under StatementUsesMemory.
- The data-flow and alias analyses take into account intra-procedural flow and move operators, as well as inter-procedural formal-actual variable assignments.
- The relations interfacing with other static analysis modules are best understood as inputs. However, the memory modeling feeds back into some of the same relations—we shall see constant-folding as an example.
- The rules employ symbol concatenation (++) and arithmetic (+/-) in order to create fresh *values* (arising during the analysis). Creating values is outside the standard model of Datalog evaluation, however it is supported by all realistic implementations. Notably, symbol concatenation is a native operation in the Soufflé Datalog engine [Jordan et al. 2016] that we use. Arithmetic over 32-bit integers is also native in Soufflé, however arithmetic over 256-bit EVM integers is defined using a C++ library that we integrate in the engine using the engine's foreign-function interface (*FFI*). (We have argued elsewhere [Smaragdakis 2019] about the importance of escape hatches, such as FFIs, for declarative languages.)

4.3 Analysis

The analysis has as its goal to compute the output relations shown in the bottom group of Figure 2. We do a rule-by-rule walkthrough to aid understanding and offer commentary.

The analysis begins from processing the statements matching program idioms that load the free-memory pointer value. Symbolic values based on the free-memory pointer are created through symbol concatenation (++) of a statement identifier and a 0 offset.

```
Variable_Value+(to, val),
FreePointerBasedValue(val, mload, 0) :-
  mload: [to := MLOAD(FreePtr)], val = mload ++ "0x0".
```

Arithmetic over free-pointer-based symbolic values can easily be defined. For instance, addition of a constant produces a new free-pointer-based value, per the rule below.

```
Variable_Value+(to, val),
FreePointerBasedValue(val, mload, res) :-
  [to := ADD(numVar, freePtrBasedVar)],
  Variable_Value(numVar, numVal1),
  Variable_Value+(freePtrBasedVar, freePtrBasedVal),
  FreePointerBasedValue(freePtrBasedVal, mload, numVal2),
  res = numVal1 + numVal2,
  val = mload ++ numVal1 + numVal2.
```

Symbolic subtraction of two free-pointer-based values yields a constant (and feeding back to the constant-folding analysis). FreePointerDiff computes the difference of two symbolic expressions offsetting the free-memory pointer by constants, while ensuring that the free-memory pointer has not changed value between the two instructions. This is an important predicate in later inferences.

```
FreePointerDiff(freePtrBasedVal1, freePtrBasedVal1, diff) :-
  FreePointerBasedValue(freePtrBasedVal1, mload1, numVal1),
  FreePointerBasedValue(freePtrBasedVal2, mload2, numVal2),
  UnchangedFreePointer(mload1, mload2),
  diff = numVal1 − numVal2.

Variable_Value(to, diff) :-
  [to := SUB(var1, var2)],
  Variable_Value+(var1, freePtrBasedVal1),
  Variable_Value+(var2, freePtrBasedVal2),
  FreePointerDiff(freePtrBasedVal1, freePtrBasedVal1, diff).
```

We can now see the first inference pattern. It combines information from several code sites to increase confidence in its reverse-engineering inference.

```
IsArrayVar(array),
Array_ElementSize(array, elementSize) :-
  [array := MLOAD(FreePtr)],
  [MSTORE(array, arrayLengthVar)],
  [arraySizeBytes := MUL(arrayLengthVar, sizeVar)],
  Variable_Value(sizeVar, elementSize),
  [arrayDataStart := ADD(array, 32)],
  [updatedFMP := ADD(arraySizeBytes, arrayDataStart)],
  [MSTORE(FreePtr, updatedFMP)].
```

In words: an address read from the free-memory pointer is confirmed to be an array, with elements of size *elementSize*, if there is a store to its beginning and the stored variable appears to be the array length (i.e., number of elements), as corroborated by several other pieces of evidence: the length is multiplied by another variable holding a constant (the element size) to yield the array size in bytes; the array address gets added 32 bytes (the size of the array length word) to produce the array data start address; the data start address plus the array size in bytes is stored to the free-memory pointer address.

(Note that, for reasons of conciseness, we omit the statement identifiers for all of the above arithmetic and `MSTORE` instructions. In the actual implementation, these are occasionally used in further filters, e.g., dictating which instructions can reach which others.)

The above is not the only pattern for detecting arrays.

```
IsArrayVar(array) :-
  [array := MLOAD(FreePtr)],
  Alias(array, arrayAlias),
  [arrayDataStart := ADD(arrayAlias, 32)],
  StatementUsesMemory(stmt, arrayDataStart, lengthVar),
  !Variable_Value(lengthVar, _).
```

That is, a hint that an address corresponds to an array (yet without knowing its element size or length) is when a memory-using statement is applied to an offset of 32 bytes from the address (modulo aliasing of variables) and the length of the data accessed is not a constant.

Another source for inferring arrays with known element size (1 byte) is when encountering `CALLDATALOAD` and `CALLDATACOPY` instructions used together. The former reads a word of data from the call-arguments area, the latter copies a requested number of bytes from that area to an in-memory array.

```
IsArrayVar(arrayVar),
Array_ElementSize(arrayVar, 1) :-
  [toVar := CALLDATALOAD(argIdVar)],
  Variable_Value(argIdVar, _),
  [plusFourVar := ADD(toVar, 4)],
  [lenVar := CALLDATALOAD(plusFourVar)],
  [MSTORE(arrayVar, lenVar)],
  [CALLDATACOPY(arrayDataStart, _, lenVar)],
  [arrayDataStart := ADD(arrayVar, 32)].
```

This seemingly complex rule decodes arguments based on the ABI conventions. It first recognizes a `CALLDATALOAD` instruction that reads a constant argument index (i.e., the *n*-th parameter passed from the contract's caller), interprets the contents (*toVar*) of this argument as another offset into the call-arguments area and performs another `CALLDATALOAD` after adding 4 bytes to the offset. (The call arguments start with a 4-byte encoding of the called method and the offset encoded inside arguments needs to be adjusted for these 4 bytes.) This indirection follows the ABI convention for passing arguments of non-fixed length into a contract. The second `CALLDATALOAD` is to the start of an array supplied as an argument and, hence, reads its length word. A memory store of that length to a memory address indicates the array start and this, when corroborated by a `CALLDATACOPY` to the array start address plus 32 bytes confirms the pattern: the code is reading a byte array into memory.

Note the contrast between the last two rules. The first only infers IsArrayVar without full information on the array length or element size. Yet it requires heavier analysis machinery than the second, building on non-local aliasing results. In contrast, the second rule infers high-precision information, via an intricate pattern of ABI conventions, yet requires nearly no analysis sophistication: all variables used are local, with each instruction directly feeding the next (without any Alias or Flows inferences, nor symbolic reasoning sophistication). The interplay of definite and indefinite patterns is a feature of the memory-modeling analysis. Information is gleaned in multiple ways, from all possible sources, using inferences that range from local pattern matching to remote flow to constant folding and symbolic arithmetic. Having a unified framework for combining disparate information sources (such as the one provided by a Datalog knowledge-base model) is essential to the power of the analysis.

With array addresses recognized reliably, the next element of the analysis is inferring array access patterns.

```
ArrayIndexAccess(array, indexVar, index) :-
  Array_ElementSize(array, elementSize),
  [indexVar := ADD(array, var)],
  [var := ADD(other, 32)],
  [other := MUL(index, sizeVar)],
  Variable_Value(sizeVar, elementSize).

ArrayLoadAtIndex(mload, REP(array), index, to) :-
  ArrayIndexAccess(array, indexVar, index),
  mload:[to := MLOAD(indexVar)].

ArrayStoreAtIndex(mstore, REP(array), index, from) :-
  ArrayIndexAccess(array, indexVar, index),
  mstore:[MSTORE(indexVar, from)].
```

The first of the above rules recognizes array accesses at a precisely modeled index. It checks that the code is adding a variable *var* to an array base address, where *var* is 32 plus a multiple of the array element size. (Recall that the array base address points to the 32 bytes of the array length

word, followed by the array data.) This relation is then leveraged to infer array load and store operations. The **REP**(*array*) function is a syntactic shorthand for illustration purposes: it denotes a unique representative among all array variables that are inferred to hold the same array value.

Lastly, when we cannot match a specific indexing pattern, we can infer an array load or store operation with lower precision, with no index expression.

```
ArrayLoad(mload, REP(array), to) :-
  mload: [to := MLOAD(indexVar)],
  Flows(arrayDataStart, indexVar),
  Variable_Value+(array, arrayVal),
  Variable_Value+(arrayDataStart, dataStartVal),
  FreePointerDiff(arrayVal, dataStartVal, 32),
  IsArrayVar(array),
  !ArrayLoadAtIndex(mload, _, _, _).

ArrayStore(mstore, REP(array), from) :-
  mstore: [MSTORE(indexVar, from)],
  Flows(arrayDataStart, indexVar),
  Variable_Value+(array, arrayVal),
  Variable_Value+(arrayDataStart, dataStartVal),
  FreePointerDiff(arrayVal, dataStartVal, 32),
  IsArrayVar(array),
  !ArrayStoreAtIndex(mstore, _, _, _).
```

Note the use of predicates Flows and FreePointerDiff in the above rules. Even though the *indexVar* may be derived from *arrayDataStart* through arbitrary arithmetic, we can still check that the values are data-flow related through Flows. Furthermore, although *array* and *arrayDataStart* are unknown symbolic addresses (based on the free-memory pointer, which also has not changed in the meantime), their difference is a known constant, as ensured by FreePointerDiff. The combination of these elements lends significant precision to the analysis.

Exporting high-level results. The above rules conclude the presentation of a representative core of memory modeling inferences. It is instructive to see how these inferences are then used by high-level analyses.

The first output of the analysis is the above relations ArrayStore/ArrayStoreAtIndex and ArrayLoad/ArrayLoadAtIndex, which lift low-level operations to operations in array. Most of the use of memory is not due to explicit arrays in the source program, however. Instead, memory-consuming statements (`SHA3`, `CALL/DELEGATECALL`, `LOG`, `RETURN`, `REVERT`, and more) are associated with their real source-program arguments, via a relation MemoryStatement_ActualArg(*stmt, i, actual*): the statement has its *i*-th high-level argument be a memory array, whose address is held in the *actual* intermediate-language variable. Note that in the low-level (bytecode) form of the smart contract, this high-level argument may be many tens of instructions away and the apparent argument of the memory-consuming statement is instead a memory address and length, which are now completely omitted.

This lifting is performed via relatively straightforward rules. The first rule below introduces an intermediate relation that combines a memory-using statement of statically-calculated length with all sources that contribute to its memory buffer, while computing their constant memory offset. The final output (second rule) merely orders these offsets and assigns an ordinal *i* to each of the sources.

```
StatementUsesMemoryAtIndex(stmt, relativeIndex, actual) :-
  StatementUsesMemory(stmt, startVar, lenVar),
  Variable_Value+(startVar, startVal),
  Variable_Value(lenVar, lenVal),
  MatchingMSTORE(mstore, stmt),
  mstore: [MSTORE(indexVar, actual)],
  Variable_Value+(indexVar, indexVal),
  FreePointerDiff(indexVal, startVal, relativeIndex),
  lenVal > relativeIndex >= 0.

MemoryStatement_ActualArg(stmt, i, actual) :-
  order(StatementUsesMemoryAtIndex(stmt, _, actual)) = i.
```

Via such lifting, the vast majority of `MSTORE`/`MLOAD` statements (95% and 93% respectively, in our extensive experimental data set) can be replaced by high-level operations over identified non-fixed-length arguments. This brings numerous advantages to the precision of further analyses.

Stability. Many of the patterns of our memory modeling may look compiler-specific. They are, however, either impossible or highly unlikely to change in the future. The ABI, standardizing the passing of arguments in public calls, events, and returns, is considered stable. Incompatible changes in it will break communication (via external calls) with currently deployed contracts. Furthermore, argument inference of memory operations on data of statically-computable length (for example, the earlier rule producing StatementUsesMemoryAtIndex facts), over memory offsets that are either constant or free-memory-pointer-based, will remain effective as long as the underlying value-flow component can populate the values of the length variables.

Additionally, very popular and significant Ethereum infrastructure (e.g., the OpenZeppelin ECDSA library) uses inline assembly that relies on how arrays are structured in memory. This suggests that the Solidity compiler convention for encoding arrays will also be fairly stable (or painful to update). Given the stability of the layout of dynamically-sized arrays in memory, our patterns regarding their inference (for example the three rules producing IsArrayVar facts) and use (consider the rules producing ArrayStore, ArrayLoad, ArrayStoreAtIndex, ArrayLoadAtIndex) will remain valid. There may conceivably be some fragility in the translation of language features in the future, but it should be easy to capture as new rules with the same overall machinery: the interplay of concrete syntactic patterns and flow/alias/interval/constant analysis.

Generality. Since Ethereum is the dominant programmable blockchain, its low-level language (EVM bytecode) has been adopted by other major, high-profile blockchains, such as Tron [Various 2018] and Quorum [ConsenSys 2020]. Other major blockchain technologies, such as Hyperledger [Various 2015], have EVM integrations, allowing smart contracts to be written and compiled using the Ethereum toolchain (i.e., producing EVM bytecode for deployment). Therefore, the concepts and techniques we describe are currently highly relevant to the majority of the smart contracts universe. In the not-so-immediate future, Ethereum 2.0 may switch to eWASM as its virtual machine. The transition retains the concept of "memory" that costs gas to extend [Various 2019]. It is, therefore, likely that if eWASM becomes the low-level IR for Ethereum, the translation of high-level programs to it will have very similar characteristics (e.g., a free-memory pointer) to those explored in our work.

Vyper Compatibility. Solidity is the dominant (by far) EVM language, but it is not the only one. Vyper is the second most-dominant language and the Vyper compiler uses EVM memory in notably different ways. The main difference is that the Vyper compiler is more aggressive in its memory use, placing nearly all local variables in memory (instead of the EVM stack, as the Solidity compiler

does). Local variables have statically-known memory offsets: Vyper does not allow intra-contract recursion! This is a strictly easier case than having memory indexes relative to a free-memory pointer, hence our approach directly applies. In fact, because Solidity also uses constant indexes to a limited extent, our current implementation covers the recognition of high-level uses through constant memory in the same way as it does for free-memory-pointer-based ones. However, there is an engineering complication, in that memory modeling for Vyper (although conceptually simpler) needs to be performed earlier in the decompilation pipeline for full code coverage: the presence of local variables, as well as the control-flow graph (i.e., jump target values) are determined by data read through memory.

5 CLIENT ANALYSES

We next introduce three practical, modern clients that demonstrate the benefits of precise memory modeling for demanding second-generation Ethereum static analysis. All clients derive from recent experience and developments in the Ethereum community. We begin with more general background on how the modeling of external calls requires our earlier memory model. The client analyses described in this paper are deployed [7] (together with several more analyses) over all Ethereum-based blockchains at contract-library.com, with results updated in quasi-real time.

5.1 Background: CALLs and Memory Modeling

Two of the client analyses we will present, the *tainted ERC20 token* `transfer` analysis and the *repeated calls* analysis, rely on the extraction of high-level information from low-level `CALL` instructions. Such analyses require memory modeling, as both passing the arguments to an external call and getting its return value back are done through non-trivial uses of memory (i.e., not at constant locations, but relative to the free-memory pointer, and possibly of unbounded size). Even recognizing the target method of a call relies on modeling memory, as the first four bytes on the *calldata* (external-call input data segment) contain the *function-selector*, the first four bytes of the keccak-256 hash of the target function's signature. The return data of the external call are usually (when statically-sized) placed directly in the caller's memory buffer. When the return data are dynamically-sized, they are copied from the return-data memory segment using the `RETURNDATACOPY` instruction.

As an example, for the high-level Solidity call:

```
callee.transfer(address(0xa359c308C72B7aaf819D8F98346040EC1257ECf9), 1000)
```

The *calldata* of the internal EVM transaction will be:

```
a9 05 9c bb 00 00 00 00 00 00 00 00 00 00 00 00 a3 59 c3 08 c7 2b 7a af 81 9d 8f 98 34 60 40 ec
12 57 ec f9 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 03 e8
```

With the first four bytes being the first four bytes of `a9059cbb2ab09eb219583f4a59a5d0623ade346d962bcd4e46b11da047c9049b`—the keccak-256 hash of `"transfer(address,uint256)"`—and the rest being the two argument values padded to 32 bytes.

The code snippet below provides a simplified view of our IR for the same high-level call. Expressions relative to the free-memory pointer can be easily discerned.

```
1 Block 0x74:
2   va1 = MLOAD[FreePtr]
3   MSTORE[va1] =  0xa9059cbb << 224
```

[7] As of mid-October 2020, the corresponding contract-library warning identfiers for the 3 analyses presented in subsections 5.2, 5.3, and 5.4 are "Tainted ERC20 Token Transfer", "Fallback Will Fail", and "Repeated Calls", respectively. (For "Repeated Calls" the results corresponding to the analysis configuration we present are the ones annotated with "high confidence" on contract-library.)

```
4    va7 =  va1 + 4
5    MSTORE[va7] = 0xa359c308c72b7aaf819d8f98346040ec1257ecf9
6    vda = va7 + 32
7    MSTORE[vda] = 1000
8    ve0 = vda + 32
9
10   ve9 = MLOAD[FreePtr]
11   vec = ve0 - ve9
12
13   vf2 = EXTCODESIZE vcallee
14   vf3 = ISZERO vf2
15   vf5 = ISZERO vf3
16   JUMPI vf6(0xfe) vf5
17
18 Block 0xfa:
19   REVERT
20
21 Block 0xfe:
22   v101 = vcallee.CALL {args:MEM[ve9 len vec], return:MEM[ve9 len 32], value:0,
         gas:GAS} ...
```

Our analysis is able to infer that the MSTORE instructions at lines 3, 5, and 7 write the 0th, 1st, and 2nd arguments of the CALL instruction at line 22 to memory. We use the value flowing to the 0th argument to identify the target method of a CALL instruction. Our modeling of the Ethereum ABI allows us to also detect CALL arguments of non-fixed length.

5.2 Client: Taint Analysis

A precise memory modeling can be a building block of a taint analysis aiming to track information flow through memory operations in a precise and complete manner. We use our memory modeling to add a new client analysis to the recent Ethainter tool [Brent et al. 2020]. Additionally, we added more guard conditions to the existing Ethainter analyses. The new client analysis and guard conditions require memory modeling to be precisely expressible, since the pattern checked relies on external calls on specific method signatures, with specific arguments—all of them elements of non-constant number and size that are passed through memory.

5.2.1 Tainted ERC20 Token `transfer`. Smart contracts often hold assets other than ETH, in other tokens built on top of the Ethereum blockchain. The ERC20 standard provides a common interface for smart contracts implementing tokens on the Ethereum platform, so that token exchange and accounting can be done uniformly.

The `transfer(address recipient, uint256 amount)` function is a part of the ERC20 interface and when called transfers `amount` tokens from the caller's balance to the `recipient`'s. Contracts making external calls to the `transfer()` function usually employ guards in order to prevent untrusted entities from draining their funds. The *tainted* ERC20 token `transfer` vulnerability allows an attacker to reach an external `transfer` call in a vulnerable contract, transfering some of its tokens to an untrusted address.

```
1  contract Victim {
2    address owner;
3    function init() public {
4      owner = msg.sender;   ...
5    }
6    function withdrawTokens(address _tokenContract) public returns (bool) {
7      require(msg.sender == owner);
```

```
8       Token token = Token(_tokenContract);
9       uint256 amount = token.balanceOf(address(this));
10      return token.transfer(owner, amount);
11    }
12  }
```

The above contract provides an example of a composite *tainted* ERC20 token `transfer` vulnerability. The `withdrawTokens()` public function provides a way for the contract owner to withdraw the contract's funds in a provided token. The guard on line 8 ensures that only the owner can succesfully call this function. However the `init()` function is set to public (presumably by mistake—the most common instance is that the programmer thought they were writing a constructor but introduced a public function instead), allowing anyone to take over the contract's ownership and bypass the guard to the `transfer` call, also controlling the `recipient` address.

Our client analysis uses memory modeling to detect the real arguments of a `transfer` operation, and determine whether they can be tainted.

5.2.2 Introduction of new Guard Conditions. In addition to a new taint pattern, we improve the existing taint analyses of the Ethainter tool. This is not expected to be a high-payoff task: the improvement of our memory modeling is in increased precision, yet Brent et al. [2020] report an already very high 82.5% precision (i.e., true-positive) rate for Ethainter, obtained via manual inspection. Still, this task shows the generality of precise memory modeling: it can be used to achieve *some* benefit, even in analyses that do not primarily need good memory modeling.

Ethainter models guard conditions in Ethereum smart contracts as well as the effects their tainting can have, in order to produce composite vulnerabilities that can span multiple transactions. We introduce two new inter-contract guards that rely on precise memory modeling: both new guard conditions use an external contract as a source of authority, calling it to approve or reject an attempt to call a sensitive operation. The two patterns can be seen in the snippet below:

```
1  contract Guarded {
2    address auth = ...;
3    function sensitiveOperation() public {
4      require(msg.sender == auth.owner());
5      ...
6    }
7    function otherSensitiveOperation() public {
8      require(auth.isOwner(msg.sender));
9      ...
10   }
11 }
```

The first guard condition (showcased in `sensitiveOperation()`) can be described as follows: the contract's caller is checked against the result of an external call with no arguments to an address that is stored on the contract's storage. The second guard condition (example in `otherSensitiveOperation()`) can be described as the (boolean) result of an external call to an address that is stored in the contract's storage, taking only the contract's caller as an argument. In both cases, we consider the guarded code to be guarded by the respective storage variable, meaning that the tainting of this variable will allow an attacker to bypass the guard.

Recall that, at the low-level, the first four bytes passed through memory in an external call are the function selector bytes identifying the function to be called. These function selector bytes are present as the 0-th argument in the output (MemoryStatement_ActualArg) of our memory modeling. Similarly, the arguments sanitized by such calls are detected via memory modeling.

5.3 Client: Precise Gas Consumption

Statically estimating the maximum gas consumption for a smart contract's code is a problem of great importance and technical difficulty. Doing so with precision is also intimately tied to memory modeling. To see why, consider Figure 3, which shows the distribution of gas costs of various instruction opcodes for a large sample of Ethereum transactions.

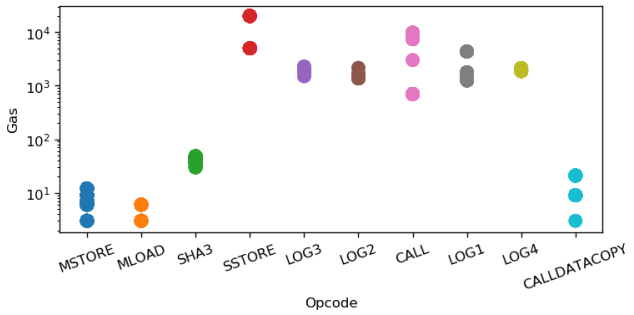

Fig. 3. Variable gas costs (logarithmic scale) for instructions operating over memory or storage. For CALL instructions, the cost shown is only that of passing arguments, not the cost of the called code.

As can be seen, the gas costs per opcode vary by orders of magnitude. Most of these instructions (MSTORE, SHA3, all LOG, CALL, CALLDATACOPY) operate over memory. Recall that instructions that use memory have a variable gas price that depends on which areas of memory are written to and how many bytes are read or written.

In a general setting (e.g., [Grech et al. 2015]), static resource consumption analysis is seen as an open problem that does not scale well to real-world applications. The cost of algorithmic code, with nested loops or loops that depend on user input, cannot easily be predicted. However, well-designed smart contracts should not contain code whose execution cost can be easily manipulated by user input. Such code would be susceptible to an unbounded operation vulnerability [Grech et al. 2018], causing the cost of executing the smart contract to skyrocket beyond hard transaction limits.

Therefore, our Ethereum gas-consumption analysis does not focus on complex loops but on highly-precise modeling of statically well-modeled loops and branches. The analysis makes use of a dynamic programming algorithm, to compute a precise (guaranteed) upper- and lower-bound gas cost. This client analysis can be instantiated with multiple configurations of gas costs to simulate alternative versions of the EVM.

The most notable practical application of the analysis has been during deliberations of the impact of Ethereum Improvement Proposal (EIP) 1884, in Aug.-Sep'19. EIP-1884 increases the cost of the SLOAD instruction from 200 to 800 units of gas. The question is to which extent deployed contracts are rendered invalid by the change: which calls made with a fixed amount of gas (not changeable by an external user) cannot complete under the newly-higher gas cost.

By far the prime example of such functionality is *fallback functions*. These are executed whenever a cryptocurrency transfer to a smart contract is performed. Solidity's `send()` and `transfer()` primitives compile into raw calls with a fixed 2,300 gas budget, in order to reduce the risk of reentrancy attacks. Under the repriced scheme, fallback functions can fail due to an out-of-gas exception.

We produced a specialized analysis to find contracts whose fallback function consume *more than 2,300 gas* with the EIP-1884 semantics but *less than 2,300 gas* with the conventional semantics. This analysis owes much of its precision to memory modeling. For instance, StatementUsesMemory was plugged into our gas cost analysis in the following rule:

```
Statement_MemoryCost(stmt, 8*len) :-
  StatementUsesMemory(stmt, _, lengthVar),
  Variable_Value(lengthVar, len).
```

Additional costs for the updated semantics are encoded as very simple rules that are picked up by the general gas-consumption client analysis.

```
Statement_EIP1884Cost(stmt, 600) :-  stmt:[SLOAD(_)].
```

5.4 Client: Repeated Calls

In Feb. 2019, during a bug bounty period for a contract in advanced development stage, a hacker drained funds [Levi 2019] from a different, collaborating contract. The attack has given rise to the *Repeated Calls* vulnerability analysis [ChainSecurity 2019], which is one of the clients of our memory modeling.

The hack, in simplified terms, consists of an untrusted party registering a new "organization" and, when asked, providing a fresh "reputation" contract. The vulnerable contract verifies that the reputation is valid but the code then asks the untrusted party for the reputation contract again, instead of remembering the previously-validated one. The attacker can provide a different reputation contract upon the repeated call, skipping validation. In the actual setting of the hack, the second contract can belong to an existing organization, allowing the draining of its funds.

This pattern gives rise to an analysis for a *bad smell* (and possibly not an outright vulnerability) in smart contract coding: making the same external call (same function, same arguments) to the same external contract address twice. Even in cases when calling the same external function twice is equivalent to calling it once, the pattern is still a bad smell because the second call wastes gas. Extra conditions to make this pattern qualify as a bad smell include that the calls return a (used) value; and that both calls are made inside the same function, i.e., will occur during the same transaction execution.

Our analysis for this pattern first checks that two calls are made to the same function—identified by the possible signature's hash—and that these calls have some return data. We also ensure that one of the calls precedes the other using a dominance analysis. We improve the precision of our analysis using the Flows and Alias analyses to ensure a common flow to both callee variables and all method-call arguments. An analysis without memory modeling would only be able to report calls that have the same callee, unable to detect the target method or arguments, which would produce very imprecise results.

6 EVALUATION

We evaluate the precise static memory modeling both in isolation, using quantitative measurements of fidelity and scalability, and as part of the client analyses that benefit from it. All analyses are evaluated on a dataset of all 70,063 unique contracts deployed on the Ethereum network up to Oct. 2019, with source code publicly available. (Having source code available enables much more productive manual inspection. If this is to introduce a bias, it is in favor of more mature contracts.) The uniqueness of these contracts is established by comparing a cryptographic hash of their bytecode. We run our static analysis tools using 24 concurrent analysis processes on an idle machine with two Intel Xeon Gold 6136 3.00GHz CPUs (each with 12 cores x 2 hardware threads, for a total of 48 hardware threads) and 640GB of RAM. We use a cutoff of 120 seconds for the static analysis. Programs that did not finish analyzing within these cutoffs are considered to have timed out. These parameters were established through experimentation. Doubling, or halving, the cutoff makes less than 0.2% difference to the total number of contracts (already the vast majority, at 99%) that our system scales to. We implement our analysis on top of the Gigahorse decompiler [Grech et al.

Table 1. Statistics for MSTORE and MLOAD instructions.

	MSTORE	MLOAD
# Total	8,700,299	5,675,818
# Modeled	8,229,559	5,262,635
% Modeled	94.59%	92.72%

2019]. The results reported in this paper can be obtained using the publicly-available peer-reviewed version of the artifact [Lagouvardos et al. 2020] accompanying the paper.

6.1 Quantitative Evaluation of Memory Modeling

Analysis Fidelity. As illustrated in Section 4, our analysis infers high-level information from low-level memory instructions. During this process we need to classify these memory-related instructions based on their function and either transform them to some high-level construct or remove them. The most common input instructions are MSTORE and MLOAD, so our evaluation focuses on these.

MSTORE instructions can be identified, or abstracted, as:

- Array allocation: Allocations of inferred arrays, which correspond to the write of the array's length in the first word of the array.
- Indexed array store: Writes to arrays modeled by relation ArrayStoreAtIndex.
- Unindexed array store: Writes to arrays modeled by relation ArrayStore.
- Update free ptr: Updates to the free-memory pointer residing in memory. This instruction on its own is not mapped to a high-level instruction.
- Argument to SHA3/LOG/CALL/...: Writes that push arguments to statements that use memory.

MLOAD instructions can be identified, or abstracted, as:

- Array length loads: Loads of the length of inferred arrays, which is stored in the first word of the array.
- Indexed array load: Reads from arrays modeled by relation ArrayLoadAtIndex.
- Unindexed array load: Reads from arrays modeled by relation ArrayLoad.
- Load free ptr: Loads of the free-memory pointer, which is stored in memory. This instruction on its own is not mapped to a high-level instruction.
- Return of call: Loads that read return data of calls.

Table 1 presents total metrics for the classification/modeling of MSTORE and MLOAD instructions while Figures 4 and 5 provide insight on the distribution of the different classes for the two instructions. Around 95% of MSTORE instructions and 93% of MLOAD instructions are modeled.

The MSTORE and MLOAD instructions that remain unmodeled are due to imprecision introduced by the decompiler or to the patterns that our rules detect breaking into different functions.

MSTORE instructions are mainly used for passing arguments to statements that consume memory. These instructions are SHA3, LOGx, RETURN, CALL, STATICCALL, DELEGATECALL, CALLCODE and REVERT. Accordingly, the majority of MLOAD instructions are loads of the free-memory pointer. This is because writing arguments of the above statements in memory usually requires one or more loads of the free-memory pointer, as this indicates where the arguments are to be written. Notice that the free-memory pointer is rarely updated, which is due to the fact that the same memory area can be rewritten and used by many memory-consuming operations.

Analysis Scalability. In order to assess the scalability of our system we ran our analysis on the dataset to see (i) how long it took to analyze the contracts and (ii) what percentage of these contracts

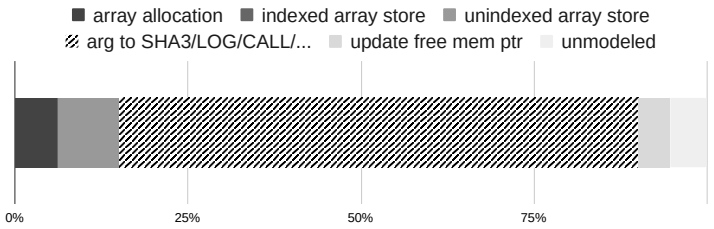

Fig. 4. Classification of MSTORE instructions

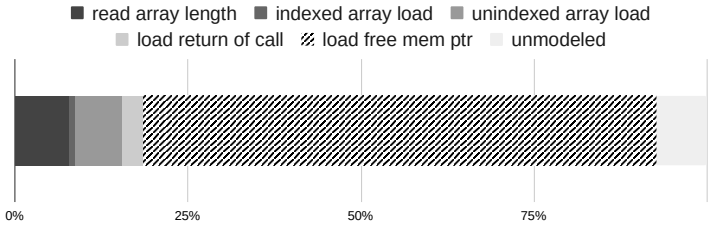

Fig. 5. Classification of MLOAD instructions

were successfully analyzed. We started off with contracts that have already been decompiled using the Gigahorse [Grech et al. 2019] decompiler (which in itself took 2 hours, with 1.01% timeouts). Our further analyses (memory modeling + clients) took an average of 2.3s of CPU time per contract. Only 4 out of these contracts timed out during the further processing, showing the negligible overhead introduced by memory modeling.

In comparison (and merely as a point of reference, since the two tools differ greatly in their architecture), the Securify tool [Tsankov et al. 2018], which we evaluate against in the repeated calls client, takes an average of 33.5s per contract (with timeouts excluded) while another 5.2% of the contracts result in a (Securify-default) timeout after 20 mins, and 3% result in an error—an average of 95s per contract with timeouts included. Both tools use the same Datalog back-end: the Soufflé engine [Jordan et al. 2016].

6.2 Client: Taint Analysis

6.2.1 Tainted ERC20 Token `transfer`. Our *tainted ERC20 token* `transfer` client reports at least one warning in 0.92% (647) of contracts in our dataset. To evaluate the precision of our analysis we performed manual inspection on 25 contracts with at least one warning reported, resulting in a 51.85% true-positive rate. The results of the manual inspection are displayed in Table 2. We consider a reported warning a true positive if it can be used to extract tokens without being authorized or having to pay ETH or other tokens. This level of precision, for a vulnerability with immediate monetary impact, makes the client analysis highly valuable. There are currently contracts holding tokens that our analysis flags and can be immediately exploited! (These contracts hold small sums, of up to a few hundred dollars. Several others flagged by our analysis have already been drained.)

6.2.2 Effects on Pre-Existing Ethainter Clients. As discussed in Section 5.2.2, we also leverage memory modeling to detect guarding patterns (and, thus, disqualify invalid warnings) for pre-existing client analyses in the Ethainter tool [Brent et al. 2020]. The existing Ethainter analyses leave very little room for precision improvement. Brent et al. report an 82.5% precision (i.e., true-positive) rate for Ethainter, obtained via manual inspection. The precision is due to the analyses being highly

Table 2. Manual inspection for the *tainted ERC20 token* `transfer` vulnerability.

MD5	LOC	TP/FP	Comment
7eacf	1441	0 / 1	requires sender to destroy token
c09fb	146	0 / 1	can only be used to send to untainted investors
cee49	244	1 / 0	composite
486df	490	0 / 1	unrecognized guard
92a49	511	0 / 1	unrecognized guard
17c8f	64	1 / 0	by design, airdrop
b1092	201	1 / 0	by design, airdrop
a4f0e	52	1 / 0	by design, airdrop
8bfbd	479	0 / 1	unrecognized guard
af93f	264	1 / 0	composite
f02a4	204	1 / 0	by design, airdrop
b30d4	1069	1 / 0	composite
78fcb	64	1 / 0	by design, airdrop
82815	652	1 / 0	composite
6ecdb	864	0 / 1	complex logic, tokens sent will be compensated
394d2	394	0 / 1	unrecognized guard
e3129	429	0 / 1	unrecognized guard
4f9ac	56	0 / 1	requires caller to transfer tokens first
29976	224	1 / 0	composite
95b19	237	1 / 0	composite
503f3	698	0 / 3	unrecognized guard
0f8ab	268	1 / 0	composite
040e6	599	0 / 1	unrecognized guard
fb0ae	227	1 / 0	composite
33350	74	1 / 0	unguarded transfer
Total:	9951	14 / 13	

tuned to common Ethereum programming patterns for guarding sensitive code. The extra patterns we add are used relatively rarely as sanitization code in realistic smart contracts.

Still, memory modeling manages to provide non-negligible benefit, showcasing the uniform precision loss that multiple Ethereum tools incur due to partial or imprecise memory modeling. Specifically, our memory modeling serves to eliminate **3%** of false positives for the "accessible self-destruct" Ethainter vulnerability, **4.5%** for "tainted self-destruct", **2.5%** for "tainted owner variable" and nearly **8%** for "unchecked tainted static call".

6.3 Client: Gas of Fallback Functions

We evaluate the precise gas analysis client on the entire corpus of smart contracts with sources. The analysis takes just under 16 minutes for 70,063 decompiled smart contracts. In order to conduct a high-fidelity experimental evaluation we developed a simple dynamic analysis tool, FallbackRetracer, specifically to automate the evaluation process of this client analysis. FallbackRetracer connects to Ethereum nodes and analyzes past Ethereum transactions to a fallback function. It can determine whether a past transaction would have failed due to an out-of-gas exception given an alternative gas semantics.

Our static analysis flags 1,760 smart contract instances (105 unique bytecodes out of 70,063) as liable to have their fallback function fail under 2,300 units of gas under the EIP-1884 costs, but

not under the original gas pricing. FallbackRetracer was used to trace the majority of historical transactions in which these flagged smart contracts were invoked.[8]

FallbackRetracer analyzed 10024 fallback function transactions, involving cryptocurrency transfers of 86K ETH, and simulated a modified gas semantics to determine whether fallback functions would fail. A summary of this experiment is shown in Figure 6.

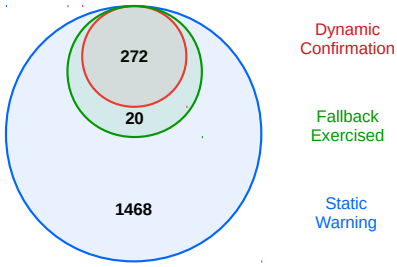

Fig. 6. Confirmation through dynamic analysis of the past-exercised static warnings, yielding 93% precision.

292 of the 1,760 flagged contracts are found to have been dynamically exercised in past transactions. Of these, FallbackRetracer confirms the correctness of the warning in 272 contracts, for a **93%** precision of our gas analysis: these contracts have had fallback function executions that did not fail under the original gas price, but would upon the EIP-1884 repricing! We inspected the remaining contracts manually and in many cases these involve branching code, where many of the branches are not exercised dynamically, indicating that the 93% precision is in fact a lower-bound. Note that an additional 1468 contracts were flagged by our analysis, but the fallback of these contracts was never exercised dynamically in past transactions. The high number validates the importance of static analysis for this question.

Examples of smart contracts that are flagged statically include the Kyber network decentralized exchange and other popular Ethereum projects, including CappedVault, Aragon OS's DepositableDelegateProxy, as well as various active token auction contracts.

The result of our analysis received publicity by the Ethereum Foundation [Swende 2019], as well as a monetary reward.

6.4 Client: Repeated Calls

Our repeated calls security analysis client flags 1,128, or 1.6%, of the 70,063 dataset of smart contracts. We compare precision to that of Securify. This is a fitting comparable, for several reasons. First, ChainSecurity, the company developing Securify, had conducted an audit of the contract vulnerable to the original repeated-calls attack [Levi 2019] and were the first to publicly report of the attack technique while adding the vulnerability analysis to Securify [ChainSecurity 2019]. Second, the results of Securify should be highly indicative of the precision without full memory modeling for this client analysis: correctly identifying the target contract, function id, and argument identities (all passed through memory) is the essence of the analysis.

Due to scalability issues with Securify we run it on a randomly selected subset of 500 contracts. 45 of the 500 contracts are flagged by Securify's bytecode *and* source analyses. (We ran the analysis

[8]Note that FallbackRetracer, or any other dynamic analysis tool, would struggle to trace *all* past transactions in Ethereum. In fact, at the time of writing, synching a full archival Geth or Parity node (the two major Ethereum implementations) requires fairly specialized hardware and over a month of runtime. It is interesting that, for this application, static program analysis is both faster and more complete than dynamic replay.

initially on bytecode for all 500 contracts and then tried to reproduce the warning on source for a smaller number, the 82-of-500 contracts flagged from bytecode, to make inspection possible. This gives Securify a precision benefit, as the final 45 warnings should be higher-confidence.)

We inspect these 45 contracts manually, and also inspect another 45 contracts randomly chosen among the ones flagged by our analysis. We show the results of manual inspection on a function-level granularity: a function is flagged if at least one false or true positive is found in it.

A flagged vulnerability is a true positive if the reported calls have the same callee and argument and can take place in the same transaction (i.e., starting from the same public function).

A complete log of the manual inspection is shown in Figures 7a and 7b. Precision (calculated as #-of-true-positives/#-of-warnings) is **16.09%** for Securify and **89.65%** for our analysis. (Recall that this is after multiple setup decisions that only increase Securify's precision.)

Comments in Figures 7a and 7b are abbreviated as follows: **NoRet**: The target function is void, or the return value is not being used; **DiffPaths**: The repeated calls that were reported cannot be in the same execution path; **DiffReceiver**: The receivers of the reported calls are different; **DiffArgs**: The actual arguments of the reported calls are different; **NoisyDecomp**: The calls are reported due to noise introduced in the decompilation phase (specific to our analysis); **Other**: Any other kind of false report.

Our false positives are often due to artifacts of imprecise decompilation. This decompilation-related imprecision can manifest itself by introducing code not available in the source to some decompiled function.

The false-positives of Securify are overwhelmingly related to memory modeling. Some common false positives are cases where the reported call-sites do not make calls to the same methods or the calls do not have return values. Another frequent cause of false positives are calls made where a loop induction variable flows to either the receiver or the arguments, making each call different.

7 RELATED WORK

Many security tools [Albert et al. 2018; Brent et al. 2018; ChainSecurity 2020; Feist et al. 2019; Grech et al. 2018; Hildenbrandt et al. 2018; Jiang et al. 2018; Kolluri et al. 2019; Krupp and Rossow 2018; Luu et al. 2016; Mavridou and Laszka 2018; Mossberg et al. 2019; Nikolić et al. 2018; Permenev et al. 2019; Tsankov et al. 2018; Various 2018] have been proposed recently aiming to verify or detect vulnerabilities in Ethereum smart contracts. Our approach for memory modeling is intended to be used in order to increase the precision and completeness of static analysis tools [Albert et al. 2018; Brent et al. 2020, 2018; Grech et al. 2018; Tsankov et al. 2018; Various 2018] operating directly on EVM bytecode.

Previous work attempted to model memory operations by resolving memory offsets to constant addresses. The state-of-the-art Securify [Tsankov et al. 2018] tool employs "partial evaluation, which propagates constant values along computations" in order to increase the amount of offsets it resolves. The ethIR [Albert et al. 2018] framework for high-level analysis of EVM bytecode also annotates `MSTORE` and `MLOAD` instructions with the memory address they operate on when it can be computed statically. Both of these approaches will fail to model completely code that uses variable-length types. For this, a static model of memory contents, based on abstract or symbolic values (such as our modeling of the free-memory pointer) is necessary. In Section 6 we compare our own analysis for the repeated calls pattern against Securify's and attribute many of its false positives to imperfect handling of memory operations.

The successor to the Securify tool, named Securify2 [ChainSecurity 2020], was released in early 2020 and performs analysis at the source level. Securify2, Slither [Feist et al. 2019] and other tools performing analysis on source code avoid the complications of analysis at the bytecode level, such as dealing with low-level memory accessses, at the cost of only being able to analyze only the

MD5	LOC	TP/FP	Comment
01400	354	1 / 0	-
f10bf	2904	1 / 0	-
61399	797	1 / 0	-
cbafe	1474	0 / 0	-
2f72d	1869	1 / 0	-
880e4	301	1 / 0	-
b6ae2	1777	2 / 1	DiffPaths
a3865	211	1 / 1	NoisyDecomp
0ac43	959	0 / 1	DiffArgs
f8cc7	31	1 / 0	-
3a7b2	1381	1 / 0	-
efbe7	449	2 / 0	-
4fea3	319	1 / 0	-
17e2a	2284	1 / 0	-
9f4cf	456	1 / 0	-
13add	627	1 / 0	-
b49ff	746	2 / 0	-
c1ce9	1765	1 / 0	-
fe966	1296	1 / 0	-
ebbb1	316	1 / 0	-
88a45	869	1 / 0	-
8dcc1	1391	1 / 0	-
f4a01	1282	1 / 0	-
f7ae2	394	1 / 0	-
947e5	2005	1 / 0	-
603f8	1381	1 / 0	-
480d1	1397	2 / 0	-
bef2d	1485	1 / 0	-
3ae0e	289	1 / 0	-
1a42f	1052	0 / 1	NoisyDecomp
c2a28	1526	1 / 0	-
369fc	443	1 / 0	-
b84dc	1559	1 / 0	-
6ef40	603	1 / 0	-
33edd	2292	1 / 0	-
138b5	724	1 / 0	-
c830a	566	1 / 0	-
2fb1a	434	3 / 0	-
5e424	1239	1 / 1	NoisyDecomp
dc6d1	2876	1 / 0	-
fade4	132	1 / 0	-
b8d52	1378	1 / 0	-
a678e	626	5 / 0	-
c115d	898	0 / 1	DiffArgs
69205	1314	1 / 0	-
Total:	48471	52 / 6	

(a) Results of our analysis.

MD5	LOC	TP/FP	Comment
01e7c	487	0 / 1	Other
12fda	309	0 / 3	DiffArgs, Other
25f64	348	0 / 1	DiffArgs
28d2b	97	0 / 1	NoRet
29806	706	1 / 3	NoRet, DiffReceiver, DiffArgs, Other
31ab8	56	0 / 1	NoRet, DiffArgs
3b141	588	0 / 1	NoRet, Other
3c8fe	1242	0 / 1	Other
3d284	459	0 / 1	DiffArgs
3e75d	889	0 / 9	NoRet, DiffArgs, Other
3f3de	550	0 / 1	NoRet
4570e	308	1 / 0	-
484fd	809	0 / 1	NoRet, DiffArgs
4aed6	2906	0 / 1	DiffReceiver
504c3	429	0 / 1	DiffArgs
51da7	273	1 / 1	NoRet, DiffReceiver, DiffArgs
52590	549	0 / 1	DiffReceiver, DiffArgs
62d9e	1782	0 / 3	DiffArgs, Other
69516	207	0 / 1	DiffReceiver
721e5	368	1 / 0	-
7d3c9	452	0 / 1	Other
7ec5d	1638	1 / 1	DiffArgs
828ee	381	1 / 1	Other
86f6a	343	0 / 1	NoRet
8c364	145	0 / 1	DiffReceiver, DiffArgs
93640	426	0 / 1	NoRet
97d07	803	0 / 2	NoRet, DiffReceiver, DiffArgs
9d622	985	0 / 4	DiffArgs, Other
9ec66	17	0 / 1	DiffReceiver, DiffArgs
a2469	273	1 / 1	NoRet, DiffArgs
a51e9	406	0 / 1	DiffReceiver
a5954	406	1 / 3	NoRet, DiffReceiver, DiffArgs
a65c5	597	0 / 2	NoRet, DiffArgs
b4e49	1288	1 / 1	Other
c3da1	972	0 / 2	NoRet, DiffArgs
c5b84	713	1 / 4	NoRet, DiffReceiver, DiffArgs
ccc15	202	0 / 2	DiffArgs
ceb6d	28	0 / 1	NoRet, DiffReceiver
d11a2	551	3 / 4	DiffArgs, Other
d58fd	3679	1 / 0	-
d5d60	143	0 / 1	Other
dd4f4	303	0 / 1	Other
e53d2	989	0 / 3	Other
f3fea	386	0 / 1	Other
f45f8	438	0 / 3	NoRet, DiffReceiver, DiffArgs
Total:	29926	14 / 73	

(b) Results of Securify's analysis.

Fig. 7. Manual inspection for the *Repeated Calls* analysis.

subset of deployed contracts that have their source code publicly available, and are written in Solidity versions supported by the source tool.

The recent Ethainter [Brent et al. 2020] tool for detecting composite information flow vulnerabilities also had an imprecise model of memory. As discussed in Section 5.2 we replaced the old memory modeling with our own and extended Ethainter with two new guard conditions and a new *tainted ERC20 token* `transfer` not possible without a precise model of EVM's memory.

We differentiate static analysis tools that attempt to model all possible states of execution from tools that make use of symbolic execution techniques [Kolluri et al. 2019; Krupp and Rossow 2018; Luu et al. 2016; Mossberg et al. 2019; Nikolić et al. 2018]. The latter have been very popular both in the literature and in practice, especially since they can more easily circumvent the low-level complexity of the EVM. However, symbolic execution approaches are inherently incomplete as they cannot model all of a program's paths.

Relatedly, several recent tools [He et al. 2019; Jiang et al. 2018; WÃijstholz and Christakis 2020] employ fuzzing techniques in order to detect vulnerabilities. Notably, WÃijstholz and Christakis [2020] use static analysis to guide a fuzzer—a combination worth exploring more.

8 CONCLUSIONS

In this paper, we argued for the importance of memory modeling for precise static analysis of Ethereum smart contracts. We presented an approach to modeling memory based on symbolic values (indexed off the free-memory pointer) and understanding of higher-level concepts (arrays, strings, ABI-encoded buffers). The analysis is aided by a combination of symbolic modeling and flow analysis and results in a precise and deep modeling of memory. This, in turn, enables or enhances the precision of valuable, modern client analyses in the Ethereum ecosystem. One such client, a precise gas modeling analysis, has already attracted attention, and will only become more practically valuable for assessing gas consumption in the future.

ACKNOWLEDGMENTS

We gratefully acknowledge funding by the Hellenic Foundation for Research and Innovation (project DEAN-BLOCK).

REFERENCES

[n. d.]. Online Solidity Decompiler. http://ethervm.io/decompile

Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio, and Ilya Sergey. 2018. EthIR: A Framework for High-Level Analysis of Ethereum Bytecode. In *Automated Technology for Verification and Analysis (ATVA)*. Springer.

Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: A Smart Contract Security Analyzer for Composite Vulnerabilities. In *Conf. on Programming Language Design and Implementation (PLDI)*. ACM.

Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A Scalable Security Analysis Framework for Smart Contracts. arXiv:cs.PL/1809.03981

ChainSecurity. 2019. Dangerous Repeated Calls to Untrusted Contracts. https://medium.com/chainsecurity/dangerous-repeated-calls-to-untrusted-contracts-3c97d614744b

ChainSecurity. 2020. Securify2. https://github.com/eth-sri/securify2

ConsenSys. 2020. ConsenSys Quorum. https://consensys.net/quorum/

Dedaub. 2019. Contract Library. https://contract-library.com/

Josselin Feist, Gustavo Greico, and Alex Groce. 2019. Slither: A Static Analysis Framework for Smart Contracts. In *Proceedings of the 2Nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB '19)*. IEEE Press, Piscataway, NJ, USA, 8–15. https://doi.org/10.1109/WETSEB.2019.00008

Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: Thorough, Declarative Decompilation of Smart Contracts. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 1176–1186. https://doi.org/10.1109/ICSE.2019.00120

Neville Grech, Kyriakos Georgiou, James Pallister, Steve Kerrison, Jeremy Morse, and Kerstin Eder. 2015. Static Analysis of Energy Consumption for LLVM IR Programs. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems (SCOPES '15)*. ACM, New York, NY, USA, 12–21. https://doi.org/10.1145/2764967.2764974

Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: Surviving Out-of-Gas Conditions in Ethereum Smart Contracts. *Proc. ACM Programming Languages* 2, OOPSLA (Nov. 2018).

Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. Learning to Fuzz from Symbolic Execution with Application to Smart Contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. ACM, New York, NY, USA, 531–548. https://doi.org/10.1145/3319535.3363230

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

Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/3238147.3238177

Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 422–430.

Tomasz Kolinko. 2018. Eveem/Panoramix – Showing Contract Sources since 2018. http://eveem.org/

Aashish Kolluri, Ivica Nikolic, Ilya Sergey, Aquinas Hobor, and Prateek Saxena. 2019. Exploiting the Laws of Order in Smart Contracts. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*. ACM, New York, NY, USA, 363–373. https://doi.org/10.1145/3293882.3330560

Johannes Krupp and Christian Rossow. 2018. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 1317–1333. https://www.usenix.org/conference/usenixsecurity18/presentation/krupp

Sifis Lagouvardos, Neville Grech, Ilias Tsatiris, and Yannis Smaragdakis. 2020. Precise Static Modeling of Ethereum "Memory" (artifact). https://doi.org/10.5281/zenodo.4059797

Adam Levi. 2019. A Technical Analysis of the Genesis Alpha Hack. https://medium.com/daostack/a-technical-analysis-of-the-genesis-alpha-hack-f8e34433c14b

Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 254–269. https://doi.org/10.1145/2976749.2978309

Anastasia Mavridou and Aron Laszka. 2018. Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach. http://aronlaszka.com/papers/mavridou2018designing.pdf

Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. *arXiv e-prints*, Article arXiv:1907.03890 (Jul 2019), arXiv:1907.03890 pages. arXiv:cs.SE/1907.03890

Bernhard Mueller. 2018. Smashing Ethereum Smart Contracts for Fun and Real Profit. https://github.com/b-mueller/smashing-smart-contracts/raw/master/smashing-smart-contracts-1of1.pdf The 9th annual HITB Security Conference.

Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18)*. ACM, New York, NY, USA, 653–663. https://doi.org/10.1145/3274694.3274743

Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2019. VerX: Safety Verification of Smart Contracts. https://files.sri.inf.ethz.ch/website/papers/sp20-verx.pdf.

Yannis Smaragdakis. 2019. Next-Paradigm Programming Languages: What Will They Look like and What Changes Will They Bring?. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2019)*. Association for Computing Machinery, New York, NY, USA, 187âĂŞ197. https://doi.org/10.1145/3359591.3359739

Martin Holst Swende. 2019. Security considerations for EIP-1884. URLomittedforanonymity,canbesearchedbasedontitle

Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 67–82. https://doi.org/10.1145/3243734.3243780

Various. 2015. Hyperledger âĂŞÂăOpen Source Blockchain Technologies. https://www.hyperledger.org/

Various. 2018. Rattle – An EVM Binary Static Analysis Framework. https://github.com/trailofbits/rattle

Various. 2018. TRON Decentralize The Web. https://tron.network/

Various. 2019. GitHub - ewasm design - metering. https://github.com/ewasm/design/blob/master/metering.md

Valentin WÃijstholz and Maria Christakis. 2020. Targeted Greybox Fuzzing with Static Lookahead Analysis. In *International Conference on Software Engineering (ICSE)*.