DOI:10.1145/3416262

MadMax: Analyzing the Out-of-Gas World of Smart Contracts

By Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis

Abstract

Ethereum is a distributed blockchain platform, serving as an ecosystem for smart contracts: full-fledged intercommunicating programs that capture the transaction logic of an account. A gas limit caps the execution of an Ethereum smart contract: instructions, when executed, consume gas, and the execution proceeds as long as gas is available.

Gas-focused vulnerabilities permit an attacker to force key contract functionality to run out of gas—effectively performing a permanent denial-of-service attack on the contract. Such vulnerabilities are among the hardest for programmers to protect against, as out-of-gas behavior may be uncommon in nonattack scenarios and reasoning about these vulnerabilities is nontrivial.

In this paper, we identify gas-focused vulnerabilities and present MadMax: a static program analysis technique that automatically detects gas-focused vulnerabilities with very high confidence. MadMax combines a smart contract decompiler and semantic queries in Datalog. Our approach captures high-level program modeling concepts (such as "dynamic data structure storage" and "safely resumable loops") and delivers high precision and scalability. MadMax analyzes the entirety of smart contracts in the Ethereum blockchain in just 10 hours and flags vulnerabilities in contracts with a monetary value in billions of dollars. Manual inspection of a sample of flagged contracts shows that 81% of the sampled warnings do indeed lead to vulnerabilities.

1. INTRODUCTION

Ethereum is a decentralized blockchain platform that can execute arbitrarily-expressive computational *smart contracts*. A smart contract can capture virtually any complex interaction, such as responding to communication from other accounts and dispensing or accepting funds. The possibilities for such programmable logic are endless. It may encode a payoff schedule, investment assumptions, interest policy, conditional trading directives, trade or payment agreements, and complex pricing. Virtually any transactional multiparty interaction is expressible without a need for intermediaries or third-party trust.

Smart contracts typically handle transactions in *Ether*, which is the native cryptocurrency of the Ethereum blockchain with a current market capitalization in tens of billions of dollars. Smart contracts (as opposed to noncomputational "wallets") hold a considerable portion of the total Ether available in circulation, which makes them ripe targets for attackers. Hence, developers and auditors have a strong incentive to make extensive use of various tools and programming techniques that minimize the risk of their contract being attacked.

Analysis and verification of smart contracts are, therefore, high-value tasks, possibly more so than in any other application domain. The combination of monetary value and public availability makes the early detection of vulnerabilities a task of paramount importance.

A broad family of contract vulnerabilities concerns *out-of-gas* behavior. Gas is the fuel of computation in Ethereum. Due to the massively replicated execution platform, wasting the resources of others is prevented by charging users for running a contract. Each executed instruction costs gas, which is traded with the Ether cryptocurrency. As a user pays gas upfront, a transaction's computation may exceed its allotted amount of gas. In that case, the Ethereum Virtual Machine (EVM), which is the runtime environment for compiled smart contracts, raises an out-of-gas exception and aborts the transaction. *A contract is at risk for a gas-focused vulnerability if it has not anticipated (or otherwise does not correctly handle) the possible abortion of a transaction due to out-of-gas conditions*. A vulnerable smart contract may be blocked forever due to the incorrect handling of out-of-gas conditions: re-executing the contract's function will fail to make progress, re-yielding out-of-gas exceptions, indefinitely. Thus, although an attacker cannot directly appropriate funds, they can cause damage to the contract, locking its balance away in what is, effectively, a denial-of-service attack. Such attacks may benefit an attacker in indirect ways—for example, harming competitors or the ecosystem, amassing fame in a black-hat community, or blackmailing.

In this work, we present MadMax:[1] a static program analysis framework for detecting gas-focused vulnerabilities in smart contracts. MadMax is a static analysis pipeline consisting of a decompiler (from low-level EVM bytecode to a structured intermediate language) and a logic-based analysis specification. MadMax is highly efficient and effective: it analyzes the whole Ethereum blockchain in just 10 hours and reports numerous vulnerable contracts holding a total value exceeding $2.8B, with high precision, as determined from a random sample.

[1] Available at: https://github.com/nevillegrech/MadMax.

The original version of this paper appeared in *Proceedings of the ACM Programming Languages* 2 (OOPSLA) (Nov. 2018).

MadMax is unique in the landscape of smart contract analyzers and verifiers. It is an approach employing cutting-edge *declarative static analysis* techniques (e.g., context-sensitive flow analysis and memory layout modeling for data structures), whereas past analyzers have primarily focused on lightweight static analysis, on symbolic execution, or on full-fledged verification for functional correctness. As MadMax demonstrates, static program analysis offers a unique combination of advantages: very high scalability (applying to the entire blockchain) and high coverage of potential vulnerabilities. Additionally, MadMax is raising the level of abstraction of automated security analysis, by encoding complex properties (such as "safely resumable loop" or "storage whose increase is caused by public calls"), which, in turn, allow detecting vulnerabilities that span multiple transactions.

2. BACKGROUND

A blockchain is a shared, transparent distributed ledger of transactions that is secured using cryptography. One can think of a blockchain as a long and ever-growing list of blocks, each encoding a sequence of individual transactions, always available for inspection and safe from tampering. Each block contains a cryptographic signature of its previous block. Thus, no previous block can be changed or rejected without also rejecting all its successors. Peers/miners run a mining client for separately maintaining the current version of the blockchain. Each of the peers considers the longest valid chain starting from a *genesis* block to be the accepted version of the blockchain. To encourage transaction validation by all peers and discourage wasted or misleading work, a blockchain protocol typically combines two factors: an incentive that is given as a reward to peers successfully performing validation, and a proof-of-work, requiring costly computation to produce a block. To see how distributed consensus and permanent record-keeping arise, consider a malicious client who tries to double-spend a certain amount. The client may propagate conflicting transactions (e.g., paying sellers *A* and *B*) to different parts of the network. As different peers become aware of the two versions of the truth, a majority will arise, because the peers will build further blocks over the version they perceived as current. Thus, a majority will soon accept one of the two spending transactions as authoritative and will reject the other. The minority has to follow suit, or its further participation in growing the blockchain will also be invalidated: the rest of the peers will disregard any of the blocks not resulting in the longest chain.

Using this approach, a blockchain can serve to coordinate all multiparty interactions with trust arising from the majority of peers, instead of being given to an authority by default.

The original blockchain, at least in its popular form, is due to the Bitcoin platform.[11] Bitcoin is explicitly a special-purpose cryptocurrency platform. Therefore, the data registered on the Bitcoin ledger can be seen as transaction parties and amounts (with minor logic permitted for cryptographic authentication). In contrast, the blockchain formulation we are interested in is the one popularized by the Ethereum platform[4, 21]: registered accounts may contain smart contracts, that is, full-fledged programs that can perform arbitrary computations, enabling the encoding of complex logic.

Ethereum smart contract programming is most commonly done in the Solidity language.[18] Solidity is a JavaScript-like language, enhanced with static types, contracts as a class-like encapsulation construct, contract inheritance, and numerous other features.

The Solidity (or other high-level language) level of abstraction is significantly removed from that of the code that directly runs on the Ethereum blockchain. Instead, Ethereum natively supports a low-level bytecode language—the Ethereum platform is essentially a distributed, replicated virtual machine, called the *Ethereum VM (EVM)*. The EVM is a low-level stack-machine with an instruction set such as standard arithmetic instructions, basic cryptography primitives (mainly cryptographic hashing), primitives for identifying contracts and calling out to different contracts (based on cryptographic signatures), exception-related instructions, and primitives for gas computation. Data is stored either on the blockchain (a memory area called *storage*), in the form of persistent data structures, or in contract-local transient *memory*.

In our work, we focus on analyzing smart contracts at the bytecode level. This is a *high-cost* design decision (due to the low-level nature of the bytecode). At the same time, the EVM bytecode level of abstraction yields a *high payoff* for analyses that target it. A bytecode-level analysis does not require a contract's source, allowing the analysis of both new and deployed contracts, originally written in any language. At the bytecode level, the input code is normalized, with all control flow being explicit, uniform, and simplified. Furthermore, the impedance mismatch between a high-level language and the EVM bytecode is often a source of confusion and error. For instance, consider the code pattern here:

```
creditorAddresses = new address[](size);
```

This code RESULTS in iteration over all locations of an array, to set them to zero. This iteration can well run out of gas. (Such code was behind a vulnerability[1] in the GovernMental[16] smart contract, for example.) The iteration is implicit at the Solidity level but immediately apparent at the bytecode level.

3. GAS-FOCUSED VULNERABILITIES

We next identify some of the most common patterns of gas-focused vulnerabilities. We employ Solidity for illustration purposes, even though our entire analysis work is at the EVM bytecode level.

The Ethereum execution model incentivizes users to minimize the number of instructions executed, by making them pay up front for the gas required to execute a transaction. Running out of gas is common, but, in most cases, this is not catastrophic: the transaction is reverted and the end user reruns it with a higher gas budget.

However, Ethereum smart contracts can relatively easily reach a state such that there will never be enough gas to run their code. The most common reason is the block gas limit of the Ethereum network—currently at 9M units of gas, which is enough for a mere few hundred writes to storage (i.e., to the blockchain).

3.1. Unbounded mass operations

The most standard form of a gas-focused vulnerability is that of unbounded mass operations. Loops whose behavior is determined by user input could iterate too many times, exceeding the block gas limit, or becoming too economically expensive to perform. The code may not have predicted this possibility, thus failing to ensure that the contract can continue to operate as desired under these conditions. This will commonly lead to a denial of service for all transactions that must attempt to iterate the loop. Consider the contract:

```
contract NaiveBank {
 struct Account {
  address addr;
  uint balance;
 }
 Account accounts[];

 function applyInterest() returns (uint) {
  for (uint i=0; i<accounts.length; i++) {
    // apply 5 percent interest
    accounts[i].balance =
      accounts[i].balance * 105 / 100;
  }
  return accounts.length;
 }
}
```

As the number of accounts is increased, the gas requirements for executing `applyInterest` will rise. Very quickly (after a mere few hundred entries are added to `accounts`), the function will be impossible to execute without raising an out-of-gas exception: the cost of the loop's instructions exceeds the Ethereum block gas limit.

Ethereum programming safety recommendations[17] suggest that programs should avoid having to perform operations for an unbounded number of clients (instead merely enabling the clients to "pull" from the contract). However, it is easy for contracts to violate this practice, without realizing that a loop's iterations are bounded only by user-controlled quantities.

An alternative recommendation is that when loops do need to perform operations for an unbounded number of clients, the amount of gas should be checked at every iteration and the contract should "keep track of how far [it has] gone, and be able to resume from that point".[17] This pattern is complex, error-prone, and (as we determine) very uncommon in practice.

3.2. Nonisolated calls (wallet griefing)

An additional way for a contract to run into out-of-gas trouble involves invoking external functionality that may itself throw an out-of-gas exception. The first element of the problem is a call that the programmer may not have considered extensively. Such calls are typically implicit, as part of Ether transfer. Sending Ether involves calling a fallback function on the recipient's side.

It is illustrative to see the issue based on the Solidity primitives and recommended practices. In Solidity, sending Ether is performed via either the `send` or the `transfer` primitive. These have different ways to handle transfer errors. For instance, `send` returns false if sending Ether fails:

```
<address>.send(uint256) returns (bool)
```

On the other hand, `transfer` raises an error (i.e., throws an exception) if sending Ether fails.

Importantly, both the `send` and the `transfer` Solidity primitives are designed with failure in mind. Both are translated into regular calls at the EVM bytecode level, but with a limited gas budget of 2300 given to the callee. This is barely enough to allow executing some logging code on the recipient's side. Therefore, the emphasis is placed on the error handling.

A good practice locally (and also used in recommended Ethereum security code patterns[17]) is using the `send` primitive always with a check of the result and aborting the transaction by throwing an exception, if a send fails. This effectively turns a `send` into a `transfer` plus any other code the user wants.

The problem arises when that exception is thrown in the middle of a loop, which is also handling other external accounts. The contract programmer or auditor may easily miss the potential threat. For instance, the loop may iterate only a bounded number of times (e.g., a contest may award money to the three leaders of a scoreboard) tricking the programmer into thinking that its gas consumption is fixed. Furthermore, it is counter-intuitive to consider that an external party will purposely abort the very transaction that gives it money. Finally, the usually-conservative naïve error handling of eagerly aborting the transaction conspires to cause the problem.

We can see the issue in example code for a vulnerability[20] appealingly termed *wallet griefing*.[2] Consider a simple loop that tries to reward the three winners of a contest:

```
for (uint i = 0; i < 3; i++)
  if (!(winners[i].send(reward))) throw;
```

The problem is that the `send` command will also result in the callback function of the winner being executed. All it takes for the contract to be vulnerable is for attackers to make themselves a winner and then provide a callback function that runs out of gas. The sender contract may never be able to recover from such conditions—for example, code clearing the winners may only appear after the end of the above loop.

3.3. Integer overflows

A programming error that commonly expresses itself as a gas-focused vulnerability results from possible integer overflows, often (but not exclusively) arising due to the Solidity-type inference approach. This is a separate pattern from the general attack of Section 3.1, as the iteration is not merely unbounded but literally nonterminating.

[2] The slang term "griefing" comes from the gaming community, where it is used to denote targeted destructive behavior meant to harass.

Consider the following contract:

```
contract Overflow {
 Payee payees[];

 function goOverAll() {
   for (var i = 0; i < payees.length; i++)
   { ... }
 } ...
}
```

The use of `var` induces a type inference problem. (Newer versions of Solidity statically detect this issue.) The inferred type of variable `i` is `uint8` (i.e., a byte), as the variable is initialized to 0 and `uint8` is the most precise type that can hold 0 while being compatible with all operations on `i`. Unfortunately, this means that a mere addition of 256 members to `payees` is enough to cause the loop to not terminate, quickly resulting in gas exhaustion. An attacker can exploit this vulnerability by adding fake `payees` using appropriate public functions (not shown) until the overflow is triggered.

4. DECOMPILING EVM BYTECODE

The first step of our gas-focused vulnerability analysis is a decompilation step, raising the level of abstraction from that of EVM bytecode to a structured intermediate language (IR): control-flow graphs (CFGs) over the three-address code. The decompilation step is *itself* a static analysis, as EVM bytecode is low-level: much closer to machine-specific assembly than to structured IRs (e.g., Java bytecode or.NET IL).

4.1. Challenges for EVM bytecode analysis

The EVM is a stack-based low-level IR with minimal structured language characteristics. In the bytecode form of a smart contract, symbolic information has been replaced by numeric constants, functions have been fused together, and control flow is hard to reconstruct. To illustrate, compare the EVM bytecode language to the best-known bytecode language: Java (JVM) bytecode—a much higher-level IR. The design differences include the following:

- Unlike JVM bytecode, EVM does not have structs, classes, or objects, nor does it have a concept of methods.
- Java bytecode is a typed bytecode, whereas EVM bytecode is not.
- In JVM bytecode, the stack depth is fixed under different control flow paths: execution cannot get to the same program point with different stack sizes. In EVM bytecode, no such guarantee exists.
- All control-flow edges (i.e., jumps) in EVM bytecode are to variables, not constants. The destination of a jump is a value that is read from the stack. Therefore, a value-flow analysis is necessary even to determine the connectivity of basic blocks. In contrast, JVM bytecode has a clearly-defined set of targets of every jump, independent of value flow (i.e., independent of stack contents).
- JVM bytecode has defined method invocation and return instructions. In EVM bytecode, although calls to outside a smart contract are identifiable, function calls inside a contract get translated to just jumps (to variable destinations, per the above point). All functions of a contract are fused in one, with low-level jumps as the means to transfer control.

To call an intracontract function, the code pushes a return address to the stack, pushes arguments, pushes the destination block's identifier (a hash), and performs a jump (which pops the top stack element, to use it as a jump destination). To return, the code pops the caller basic block's identifier from the stack and jumps to it.

4.2. Decompilation approach

MadMax was originally based on the Vandal decompiler.[3, 19] Subsequently, the same analysis logic has been ported to our Gigahorse decompiler framework.[6]

Our decompilation step accepts EVM bytecode as input and produces output in a standard structured intermediate representation: a control-flow graph (of basic blocks and the edges connecting them); three-address code for all operations (instead of operations acting on the stack); and recognized (likely) function boundaries. This representation is encoded as relations (i.e., tables) and queried, recursively, to formulate higher-level program analyses.

We observe that the EVM bytecode input is much like a functional language in continuation-passing-style (CPS) form: all calls and returns are forward calls (jumps), where calls add the continuation (return-to instruction) as one of the arguments. This equivalence of CPS and low-level jumps has been observed before—most explicitly by Thielecke.[15]

The technical setting of having CPS input and needing to detect value and control flow is precisely that of *control-flow analysis (CFA)*.[12, 13] Control-flow analysis is also one of the original proposals for a *context-sensitive* (*call-site sensitive*) static analysis of value flow: for a *k*-CFA analysis, every call target gets analyzed separately for each caller (i.e., calling instruction), caller's caller, etc., up to a maximum depth, *k*.

Decompilation, therefore, adopts the standard form of a control-flow analysis,[13] formulated as an abstract-interpretation. Context sensitivity adapts to the complexity of the input contract, often resulting in analyses with deep context (e.g., $k = 12$). The end result is a three-address code using the schema listed in Figure 1. Syntax sugar and minor detail elision are employed for presentation purposes. Language syntax is quoted using [and] and implicitly unquoted for meta-variables. For instance, *s*:[*to*:= BINOP(*x, y*)] indicates that statement *s* is some binary operation on *x* and *y* with its result in *to*, where *x*, *y*, and *to* are the meta-variables referring to the bytecode variables. The distinction between variables in the analyzed program and meta-variables in the analysis is clear from context; therefore, we simply refer to "variables," henceforth. We omit the statement identifier, *s*, when it does not affect a rule. We also use $*$ as a wildcard, that is, it denotes any variable, which is ignored.

The schema captures all elements of EVM bytecode in a slightly abstracted fashion, using a standard, structured intermediate language. For example, JUMPI instructions have statements, and not arbitrary values, as targets. All binary operations are treated equivalently, as we currently do not attempt to analyze arithmetic expressions. We do not

include unary operations or direct assignment between variables in Figure 1, although we do so in the implementation, because these can be treated as special cases of binary operations. RTVALUE gives a uniform treatment of instructions that return the cost of gas, transaction id, code size, caller, and other run-time quantities.

Figure 1. Domains and decompiler output (i.e., input relations for main analysis).

V is a set of program variables
C is a set of constants, $C \subseteq \mathbb{Z}$
S is a set of statement identifiers
$\mathbb{N}$ is the set of natural numbers, $\mathbb{Z}$ is the set of integers

constant assignment	
s:[to := CONST(c)]	$s : S, to : V, c : C$
load from storage	
s:[to := SLOAD ($index$)]	$s : S, index : V, to : V$
store to storage	
s:[SSTORE($from$, $index$)]	$s : S, index : V, from : V$
load from (volatile) memory	
s:[to := MLOAD($index$)]	$s : S, index : V, to : V$
store to (volatile) memory	
s:[MSTORE($from$, $index$)]	$s : S, index : V, from : V$
conditional jump	
s:[JUMPI($cond$, $label$)]	$s : S, cond : V, label : S$
conditional throw	
s:[THROWI($cond$)]	$s : S, cond : V$
keccak 256 hash	
s:[to := SHA3(ind, len)]	$s : S, ind : V, len : V, to : V$
call external contract	
s:[to := CALL($addr$, gas...)]	$s : S, addr : V, gas : V, to : V$
get remaining gas	
s:[to := GAS()]	$to : V$
get run-time value (e.g. current block size)	
s:[to := RTVALUE()]	$to : V$
CAST integer to a number of bits	
s:[to := CASTN($from$)]	$to : V, from : V, n : \mathbb{N}$
binary operator e.g. ϕ, ADD, AND, etc.	
s:[to := BINOP(a, b)]	$y\ s : S, a : V, b : V, to : V$

5. CORE MADMAX ANALYSIS

The main MadMax analysis operates on the output of decompilation using logic-based specifications. The analysis is implemented in the Datalog language: a logic-based language, equivalent to first-order logic with recursion.[8] The analysis consists of several layers that progressively infer higher-level concepts about the analyzed smart contract. Starting from the three-address-code representation of Figure 1, concepts such as loops, induction variables, and data flow are first recognized. Then, an analysis of memory and dynamic data structures is performed, inferring concepts such as dynamic data structures, contracts whose storage increases upon reentry, nested arrays, etc. Finally, concepts at the level of analysis for gas-focused vulnerabilities (e.g., loop with unbounded mass storage) are inferred.

5.1. Flow and loop analyses

Ethereum gas-focused vulnerabilities tend to require a high-level semantic understanding of the underlying contract. There are various initial low-level analyses that need to happen before expressing deeper semantics. Thus, the first step of a MadMax analysis is the derivation of loop and data flow information. This yields several relations, on which further analysis steps are built. The relations, together with some extra domain and input context definitions, are given in Figure 2. We do not provide the Datalog rules for any of these relations—their implementation, although not always straightforward, is standard. For instance, it resembles the flow computation in standard Datalog analysis formulations[14] or frameworks for Java bytecode, such as JChord[9, 10] and Doop.[2]

The first three computed relations in Figure 2 (INLOOP, INDUCTIONVAR, and LOOPEXITCOND) encode useful concepts in structured loops. Note that loops in low-level programs do not have to be structured; for example, there may not be a loop head that dominates all loop statements. However, Solidity and other EVM languages often produce structured loops as part of their compilation process. The loop analysis finds induction variables, that is, variables that are incremented by a predictable (but not necessarily statically known) amount in each iteration.

The next four relations capture a data-flow analysis. Relation FLOWS expresses a data-flow dependency between variables. In its simplest form, FLOWS is just the reflexive transitive closure of the BINOP input relation; that is, it ignores storage and

Figure 2. Extra domains, input, and output schema for baseline loop and data flow analyses.

F is a set of function hashes
L is a set of structured loops

INPUBLICFUNCTION($s : S, f : F$)	Statement s is part of function f
INLOOP($s : S, l : L$)	Statement s is part of loop l
INDUCTIONVAR($v : V, l : L$)	v is an induction variable of loop l
LOOPEXITCOND($condVar : V, l : L$)	Loop condition of l is captured by $condVar$
HASCONSTANTVALUE($v : V, c : C$)	Constant c may propagate to variable v
FLOWS($from : V, to : V$)	Data flow analysis: the value of $from$ flows to to
VARALIAS($v : V, u : V$)	Local alias analysis: v,u may be aliased via direct assignment
MEMCONTENTS($s : S, p : V, v : V$)	At statement s, contents at memory location p may be v

memory load and store instructions. However, one can give more sophisticated Flows definitions without affecting the rest of the analysis. VarAlias is a similar relation but more restrictive, for variables directly assigned to each other with no further arithmetic. Accordingly, HasConstantValue does a simple constant propagation: it is just the composition of VarAlias with the input CONST relation.

Finally, MemContents does a simple analysis of Mstore operations given the results of VarAlias and propagates the results to every statement reachable from an Mstore in the control-flow graph.

There are two points worth mentioning about the above relations:

- The data-flow analysis (i.e., relations HasConstantValue, Flows, VarAlias, and MemContents) is best-effort, that is, neither sound nor complete. This means that, first, not all possible flows, aliases, etc. are guaranteed to be found: two variables may hold the same value as a result of complex arithmetic, runtime operations, memory load and stores, etc., without the analysis computing this. Second, not all inferences are guaranteed to hold. For example, an inference that is known to hold in one control-flow path but not in another will be optimistically propagated when paths are merged.

  The property of being neither sound nor complete carries over to our overall analysis results. MadMax neither guarantees to detect all gas vulnerabilities nor guarantees that every gas vulnerability reported is a real bug. This design choice is well-aligned with the intended purpose of a bug-detecting static analysis—the value of the analysis is not based on its guarantees but on its real-world usefulness.[5]
- Relations Flows and VarAlias are pervasive in the MadMax analysis. Most other relations we shall see henceforth are transitively closed with respect to either Flows or (the weaker) VarAlias. We elide such transitive-closure Datalog rules from our exposition and only focus on the seed logic of each interesting concept.

Armed with the above basic loop and data-flow analyses, we can establish higher-level concepts, such as a loop's bound. This is defined as LoopBoundBy in Figure 3. If both an induction variable *i* and a noninduction variable *c* flow to a loop exit condition, then we infer that the loop may be bound by the contents of *c*. A further refinement of this relation is DynamicallyBound, which infers which loops are bound by either storage or some other value that is only known at run-time.

Finally, we define predicate PossiblyResumableLoop, to match loops that appear to implement the Ethereum secure coding recommendations,[17] by checking the amount of remaining gas, saving to (permanent) storage an induction variable, and loading the same induction variable from storage. Note that this is not an entirely precise detection of resumable loops—it may well be finding instances of code that just happen to match these abstract conditions, for example, gas check, store, and load of induction variable.

Figure 3. Inferring bound loops and resumable loops.

```
LoopBoundBy(loop, var) ⟵
  InductionVar(i, loop),
  !InductionVar(var, loop),
  Flows(var, condVar),
  Flows(i, condVar),
  LoopExitCond(condVar, loop).

DynamicallyBound(loop) ⟵
  [dynVar := SLOAD(*)],
  LoopBoundBy(loop, dynVar).
DynamicallyBound(loop) ⟵
  [dynVar := RTVALUE()],
  LoopBoundBy(loop, dynVar).

PossiblyResumableLoop(loop) ⟵
  [gas := GAS()],
  LoopBoundBy(loop, gas).
  InductionVar(i, loop),
  Flows(loaded, i),
  [loaded := SLOAD(*)],
  Flows(i, stored),
  [SSTORE(*, stored)],
```

However, the existence of all three conditions is a very strong indication that the programmer has considered the possibility of an out-of-gas exception and has taken precautions to make the loop resumable on a re-execution of the contract function.

5.2. Analysis of memory layout

A faithful modeling of the Ethereum VM memory layout for dynamic data structures is a key part of MadMax. This modeling is necessary for reducing the false-positive rate of the analysis. An intuitive but naïve approach to find gas vulnerabilities may be to flag any contract that contains loops that are "dynamically bound," or loops where the number of iterations depends on some value stored in storage or passed as external input. However, a precise analysis requires more sophistication. We find experimentally that around half of the currently deployed contracts have dynamically bound loops—but it would be entirely unrealistic to expect that half of smart contracts currently deployed are vulnerable. Instead, for loops that iterate over unbounded data (i.e., data structures), we need to determine whether the data structure could have been populated by an attacker.

The Ethereum virtual machine does not have notions of high-level data structures. Instead, operations on high-level data structures are compiled down to low-level operations on addressable storage. Solidity offers two main kinds of dynamically-sized data structures: dynamically-sized arrays and associative arrays, that is, maps. Although both arrays and maps can be dynamically resized, no mechanism exists for iterating over maps. Therefore, arrays are the primary data structure to model, in order to capture loops that iterate without bounds.

The Ethereum memory layout is highly unconventional from a traditional programming language standpoint, although perfectly reasonable if one considers the specifics of the execution environment (i.e., a segregated, 256-bit

memory space per contract, cryptographic hashing as a primitive). The main idea is that a *key* represents an array. The key is the address of the memory location holding the array's size. At the same time, the key is *hashed* to yield the address of the memory location that holds the array's contents.

Figure 4 depicts an example of storage allocation for a simple contract with two scalar variables and a two-dimensional dynamic array. Fixed-sized data structures in Solidity are stored consecutively in storage as these appear in program order, starting from offset 0. The individual elements in arrays are also stored consecutively in storage; however, the starting offset of the elements requires some calculations to be determined. Due to their unpredictable size, dynamically-sized array types use a KECCAK256 hash function (SHA3) to find the starting position of the array data. The dynamic array value itself occupies an empty slot in storage at some position *p*. For a dynamic array, this slot stores the number of elements in the array. The array data, however, is located at KECCAK256(*p*). The implementation of arrays is extended to arbitrarily-nested dynamic data structures, by recursively mapping the above implementation, necessitating a recursive analysis.

MadMax performs an analysis (elided) for modeling the memory layout and identifying dynamic data structures in smart contracts. The outputs of this analysis are shown in Figure 5. Based on these relations, we define key concepts for gas-focused analyses, as shown in Figure 6. An important concept is INCREASEDSTORAGEONPUBLICFUNCTION. Storage variables that are increased and stored in their corresponding storage slot imply that a contract's array size is increased when some public function is invoked. Moreover, we can find loops that iterate over arrays. We define ARRAYITERATOR as a loop that iterates over an array.

5.3. Top level vulnerability queries

The analysis concepts of the previous sections set up the final queries for gas-focused vulnerabilities. These are made precise by combining several distinct concepts. Figure 7 shows the final output relations of the MadMax analysis in slightly simplified (and inlined to single rule) form.

Consider, for instance, the UNBOUNDEDMASSOP logic: it examines whether an array that can grow in size as the result of a public function has contents that are loaded or stored (the FLOWS(*storeOffsetVar, index*) allows dereferencing from the beginning of the contents), inside a loop whose bound is based on the array size and that contains an induction variable that affects the address loaded or stored.

The WALLETGRIEFING query is even more precise, requiring a load from the dynamic array, flow of the loaded value to a call whose result is the condition of a throw statement. The call and the throw need to be in the same loop, which also has an induction variable that affects the address loaded.

Finally, loop overflows are conservatively asserted to be likely if the induction variable is cast to a short integer or ideally one byte. The loop has to be "dynamically bound" to be vulnerable, that is, the number of iterations is determined by some run-time value.

6. IMPACT

Our original MadMax experiments consider all smart contracts available on the Ethereum blockchain on April 9, 2018. We ran MadMax on an idle machine with an Intel Xeon E5–2687W v4 3.00 GHz and 512 GB of RAM. Due to time constraints, we set a cutoff of 20 s for decompilation—beyond that time, contracts are considered to time-out.

The contracts flagged for vulnerabilities, combined, contain 7.07 million ETH, or roughly $2.8 billion.[3] In total, there were 6.33 million contract instances deployed at the time of our blockchain scraping, produced from 91.8k unique programs. 4.1% of the contracts are flagged by MadMax as being susceptible to unbounded iteration, 0.12% to wallet griefing, and 1.2% to overflows of loop induction variables.

To estimate a false-positive rate, we manually inspected a subset of the contracts flagged. Our unbiased sampling process involves taking unique bytecode programs and selecting the first and last few contracts by block-hash order. However, a bias factor is introduced by the need to have source code available online—contracts without source code were not considered, as manual inspection of low-level bytecode is highly time-consuming and unreliable.

We select the first 13 contracts, and manual inspection reveals that 11 of these contracts indeed exhibit 13 distinct vulnerabilities, of 16 flagged, for a precision of 13/16 = 81%. The exact number is hardly important—a larger sample could have it move a few percentage points up or down. What is important is that the analysis is precise enough to yield a wealth of true vulnerability warnings. By manually inspecting the sampled contracts, we have gained important insights about the effectiveness of MadMax—presented in detail in the MadMax conference publication.[7]

The entire MadMax analysis of the 91.8k contracts took less than 10 hours, running 45 concurrent processes. Subsequent advances of the Gigahorse decompiler have brought this number down by at least a factor of 2. Decompilation currently exhibits time-outs for around 4% of the contracts, depending on the exact settings.

Note that a confirmed vulnerability in a contract does not mean that: (1) exploiting the vulnerabilities is easy or cheap or (2) the vulnerability blocks all Ether in a contract. For instance, the gas required to exploit an unbounded mass

[3] The price of ETH/USD and contract balances are both volatile quantities. To fix a reference point, all numbers given are as of April 9th, 2018 (with ETH/USD at $400.72).

Figure 4. Outputs of data structure analysis.

VARINDEXESSTORAGE(*s : S, v : V*)	Variable *v* reads or writes to storage at statement *s*
ARRAYSIZEVARIABLE(*sv : V, arrId : C, kv : V*)	Array *arrId* has its length and address read in *sv* and *kv*, respectively
ARRAYIDTOSTORAGEINDEX(*arrId : C, v : V*)	*v* holds a storage address that is part of (outermost) array *arrId*

Figure 5. Storage structure and contents (bottom) for given contract (top). sha3 is the keccak256 hash function.

```
contract Foo {
  uint i0;
  uint i1;

  uint [][]a;
  ..
}
```

address	contents
0	i0
1	i1
2	a.length
sha3(2)	a[0].length
sha3(2) + 1	a[1].length
sha3(sha3(2))	a[0][0]
sha3(sha3(2)) + 1	a[0][1]
sha3(sha3(2) + 1)	a[1][0]
sha3(sha3(2) + 1) + 1	a[1][1]

Figure 6. Datalog rules for identifying storage requirements increase in public functions.

```
IncreasedStorageOnPublicFunction(arrayId) ←
  ArraySizeVariable(sizeVar, arrayId, keyVar),
  InPublicFunction([sizeVar' := ADD(sizeVar, *)], f),
  InPublicFunction([SSTORE(keyVar, sizeVar')], f).

ArrayIterator(loop, arrayId) ←
  LoopBoundBy(loop, sizeVar),
  ArraySizeVariable(sizeVar, arrayId, *).
```

Figure 7. Top-level query for unbounded mass operations, wallet griefing, and overflow vulnerabilities.

```
UnboundedMassOp(loop) ←
  IncreasedStorageOnPublicFunction(arrayId),
  ArrayIdToStorageIndex(arrayId, storeOffsetVar),
  Flows(storeOffsetVar, index),
  VarIndexesStorage(storeOrLoadStmt, index),
  InLoop(storeOrLoadStmt, loop),
  ArrayIterator(loop, arrayId),
  InductionVar(i, loop),
  Flows(i, index),
  !PossiblyResumableLoop(loop).

WalletGriefing(loop) ←
  IncreasedStorageOnPublicFunction(arrayId),
  ArrayIdToStorageIndex(arrayId, storeOffsetVar),
  Flows(storeOffsetVar, index),
  [loadVar := SLOAD(index)],
  Flows(loadVar, target),
  InLoop([resVar := CALL(target, **)], loop),
  InLoop([THROWI(condVar)], loop),
  Flows(resVar, condVar),
  InductionVar(i, loop),
  Flows(i, index).

LoopOverflow(loop) ←
  DynamicallyBound(loop),
  [to := CASTN(from, n)], n ≤ 16,
  InductionVar(to, loop),
  InductionVar(from, loop),
  Flows(to, condVar),
  LoopExitCond(condVar, loop).
```

operation vulnerability may be costly, deterring attackers. However, this does not affect the vulnerable nature of the contract against motivated malicious actors.

7. CONCLUDING DISCUSSION

We presented MadMax, a tool for finding gas-focused vulnerabilities in Ethereum smart contracts. We identify new vulnerabilities for Ethereum smart contracts and demonstrate the first successful design of a static analysis tool at the EVM bytecode level that painstakingly decompiles and reconstructs the program's higher-level semantics. The MadMax approach utilizes best-of-breed techniques and technologies: from abstract-interpretation-based low-level analysis for decompilation to declarative program analysis techniques for higher-level analysis. Our approach is validated using all deployed smart contracts on the blockchain and demonstrates scalability and concrete effectiveness. The threat to some of these smart contracts presented by our tools is overwhelming in financial terms, especially considering the high precision of warnings in a manually-inspected sample.

Gas-focused vulnerabilities are likely to become more relevant in the foreseeable future. Gas (or a quantity like it) is fundamental in blockchain computation and is, for example, included in the design of the upcoming Facebook Libra. Computation under gas constraints requires different coding styles than in traditional programming domains—a simple linear loop over a data structure may render a contract vulnerable! This year, Ethereum's *Istanbul* update makes `SLOAD` four times more expensive, whereas making `SSTORE` cheaper. Exploiting the unbounded operation vulnerability involves many state changing operations to cause the victim to perform more state reading operations. The cost to the attacker is therefore relative to the ratio of the cost of storing against the cost of reading. Hence, this vulnerability will become cheaper to exploit. Moreover, Libra's virtual machine will have state reading operations such as `ImmBorrowField` and `ReadRef`. These will be as expensive as state writing operations `MutBorrowField` and `WriteRef`, which would make the unbounded operations' vulnerability cheaper to exploit in Libra than in Ethereum.

MadMax is the first published analysis to detect threats that require coordination across multiple transactions. This is representative of the future trends for automated security analyses: the analysis will need to account for state changes by independent transactions, long before the final attack can be perpetrated. Furthermore, future threats are likely to involve multicontract or whole-app attacks—for example, with coordination between the off-blockchain part of a decentralized application and its on-blockchain (smart contract) part. This

is a challenging next frontier for security analysis tools. In the case of MadMax, multitransaction reasoning is enabled by positing high-level properties, such as "safely resumable loop." In turn, this is made possible by the declarative nature of the analysis, which allows a concise, logical specification of complex properties. The same declarative approach may well play an important role in future scaling of analyses to multi-contract, whole-application reasoning.

Acknowledgments

This research was supported partially by the Australian Government through the Australian Research Council's Discovery Projects funding scheme (project ARC DP180104030). We gratefully acknowledge funding by the European Research Council, grants 307334 and 790340. In addition, the research work disclosed is partially funded by the REACH HIGH Scholars Programme – Post-Doctoral Grants. The grant is part-financed by the European Union, Operational Program II, Cohesion Policy 2014–2020 (Investing in human capital to create more opportunities and promote the well-being of society – European Social Fund).

References

1. Atzei, N., Bartoletti, M., Cimoli, T. A Survey of Attacks on Ethereum Smart Contracts. Technical Report. Cryptology ePrint Archive: Report 2016/1007, https://eprint.iacr.org/2016/1007, 2016.
2. Bravenboer, M., Smaragdakis, Y. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of Object Oriented Programming, Systems, Languages, and Applications*, 2009.
3. Brent, L., Jurisevic, A., Kong, M., Liu, E., Gauthier, F., Gramoli, V., Holz, R., Scholz, B. Vandal: A scalable security analysis framework for smart contracts. *CoRR*, 2018. abs/1802.08660
4. Buterin, V. A next-generation smart contract and decentralized application platform, 2013. https://github.com/ethereum/wiki/wiki/White-Paper
5. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R. Extended static checking for Java. In *Proceedings of Programming Language Design and Implementation*. (2002).
6. Grech, N., Brent, L., Scholz, B., Smaragdakis, Y. Gigahorse: Thorough, declarative decompilation of smart contracts. In *Proceedings of International Conference on Software Engineering* (*ICSE*), 2019.
7. Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B., Smaragdakis, Y. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. In *Proceedings of the ACM Programming Languages*, 2 (OOPSLA) (Nov. 2018).
8. Immerman, N. Graduate texts in computer science. *Descriptive Complexity*. Springer, 1999.
9. Naik, M. Chord: A versatile platform for program analysis. In *Programming Language Design and Implementation*, 2011. Tutorial.
10. Naik, M., Park, C., Sen, K., Gay, D. Effective static deadlock detection. In *Proceedings of International Conference on Software Engineering*, 2009.
11. Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system, 2009. https://www.bitcoin.org/bitcoin.pdf
12. Shivers, O. *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie Mellon University (May 1991).
13. Shivers, O. Higher-order control-flow analysis in retrospect: lessons learned, lessons abandoned. In *Best of PLDI 1988*. K.S. McKinley, ed. Volume 39, 2004, 257–269
14. Smaragdakis, Y., Balatsouras, G. Pointer analysis. *Found. Trends Program. Lang. 1*, 2 (2015), 1–69.
15. Thielecke, H. Continuations, functions and jumps. *ACM SIGACT News*, 30 (Jan. 1999), 33–42.
16. Various. GovernMental page. http://governmental.github.io/GovernMental/.
17. Various. Safety-ethereum wiki. https://github.com/ethereum/wiki/wiki/Safety. Accessed: 2018–04–15.
18. Various. GitHub-ethereum/solidity: The solidity contract-oriented programming language, 2018. https://github.com/ethereum/solidity
19. Various. Vandal–A static analysis framework for ethereum bytecode, 2018. https://github.com/usyd-blockchain/vandal/.
20. Vessenes, P. Ethereum griefing wallets: Send w/throw is dangerous, 2016. http://vessenes.com/ethereum-griefing-wallets-send-w-throw-considered-harmful
21. Wood, G. Ethereum: A secure decentralised generalised transaction ledger, 2014. http://gavwood.com/Paper.pdf

Neville Grech (me@nevillegrech.com), University of Athens, Greece.

Michael Kong and **Anton Jurisevic** ([mkon1090, ajur4521]@uni.sydney.edu.au), The University of Sydney, Australia.

Lexi Brent and **Bernhard Scholz** ([lexi.brent, bernhard.scholz@sydney.edu.au]), The University of Sydney, Australia.

Yannis Smaragdakis (smaragd@di.uoa.gr), University of Athens, Greece.