me@nevillegrech.com

MadMax

Surviving Out-of-Gas Conditions in Ethereum Smart Contracts

Neville Grech **(Athens/Malta)**
Michael Kong (Sydney)
Anton Jurisevic (Sydney)
Lexi Brent (Sydney)
Bernhard Scholz (Sydney)
Yannis Smaragdakis (Athens)

nevillegrech.com/2018/madmax.html

1

MadMax: Surviving Out-of-Gas Conditions in Ethereum Smart Contracts

ANONYMOUS AUTHOR(S)

Ethereum is a distributed blockchain platform, serving as an ecosystem for smart contracts: full-fledged inter-communicating programs that capture the transaction logic of an account. Unlike programs in mainstream languages, the execution of an Ethereum smart contract is restricted by a gas limit: execution proceeds as long as gas is available. Thus, gas is a valuable resource that can be manipulated by an attacker to provoke unwanted behavior in a victim's smart contract (e.g., wasting or blocking funds of said victim). Gas-focused vulnerabilities exploit undesired behavior when a contract (directly or through other interacting contracts) runs out of gas. Such vulnerabilities are among the hardest for programmers to protect against, as out-of-gas behavior may be uncommon in non-attack scenarios and reasoning about it is far from trivial.

In this paper, we classify and identify gas-focused vulnerabilities, and present MadMax: a static program analysis technique to automatically detect gas-focused vulnerabilities with very high confidence. Our approach combines a control-flow-analysis-based decompiler and declarative program-structure queries. The combined analysis captures high-level domain-specific concepts (such as "dynamic data structure storage" and "safely resumable loops") and achieves high precision and scalability. MadMax analyzes the entirety of smart contracts in the current Ethereum blockchain in just 10 hours (with decompilation timeouts in 8% of the cases) and flags contracts with a current (though highly volatile) monetary value of over $5B as vulnerable. Manual inspection of a sample of flagged contracts shows that 81% of the sampled warnings do indeed lead to vulnerabilities, which we report on in our experiment.

Additional Key Words and Phrases: Program Analysis, Smart Contracts, Security

1 INTRODUCTION

Ethereum is a decentralized blockchain platform that can execute arbitrarily-expressive computational *smart contracts*. Developers typically write smart contracts in a high-level language that is translated into immutable low-level bytecode for a persistent distributed virtual machine. Smart contracts handle transactions in *ether*, a cryptocurrency with a current market capitalization of approximately $50B. Smart contracts (as opposed to non-computational "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 is, therefore, a high-value task, possibly more so than in any other programming setting. The combination of monetary value and public availability makes the early detection of vulnerabilities of paramount importance. (Detection may occur after contract deployment. Despite the code immutability which prevents bug fixes, discovering a vulnerability before an attacker could enable a trusted party to move vulnerable funds to safety.)

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 instruction executed costs gas,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 Association for Computing Machinery.
2475-1421/2018/1-ART1 $15.00
https://doi.org/

OOPSLA '18

Terminology

Smart Contracts

- Programs running on the Ethereum Blockchain (usually transacting $$$)

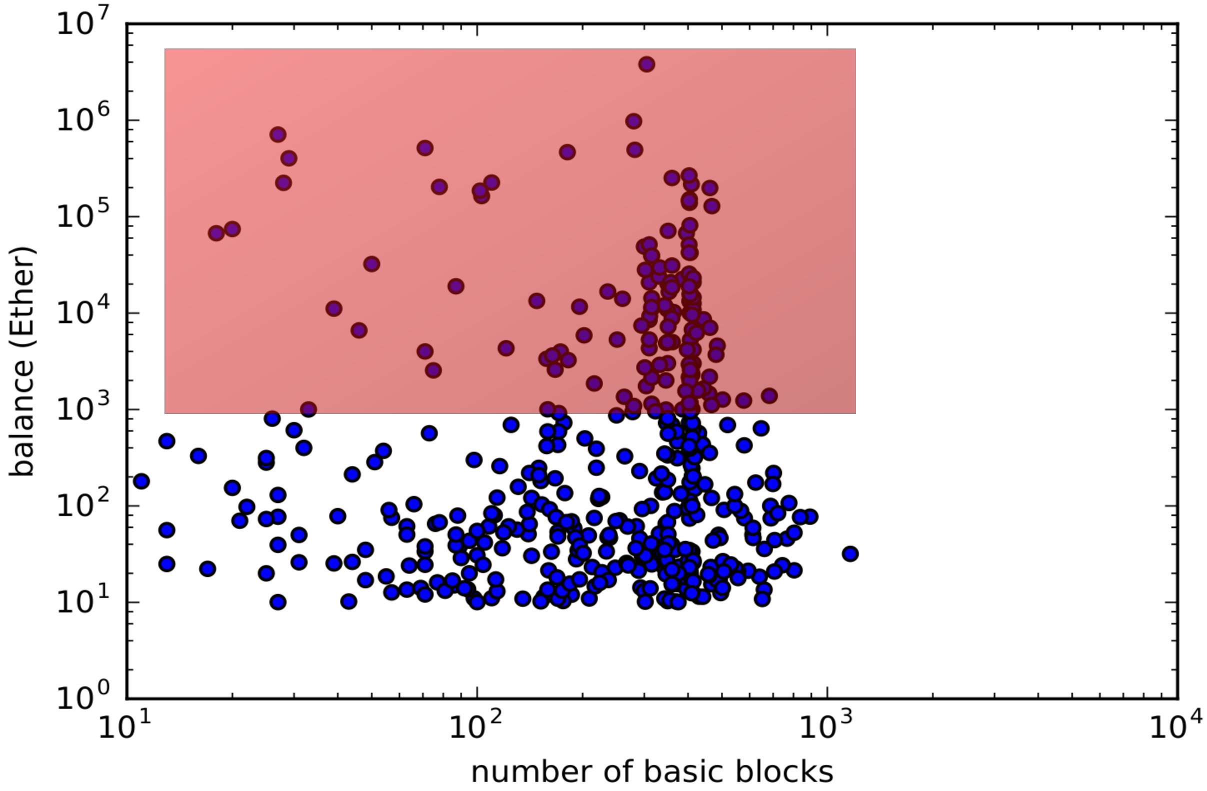
Solidity

- The high-level language for writing them

Gas

- Fee paid for running them
- Earned by the miner & bounded/hard coded

Complexity, Balance and Risk



Market cap
> $50B

Complex contracts, which hold majority of Ether, are ripe targets for attackers.

MadMax is Unique

Cutting-edge (exhaustive) *static analysis*

- Abstract Interpretation, CFA Flow Analysis, memory modeling

Performs analysis directly on the bytecode

- Source code only available for 0.34% of contracts (Etherscan)
- Developed the Vadnal decompiler for this purpose.

Evaluated on the entire Ethereum blockchain

- Found $5B on vulnerable contracts (81% estimated precision)

Gas-focussed vulnerabilities

nevillegrech.com/2018/madmax.html

Gas Focussed Vulnerabilities

- **Gas is needed to execute contracts:**
  - Paid for by the account that calls the smart contract.
  - Has monetary value - prevents wasting of resources.
  - If not enough gas is budgeted, transaction is reverted.
  - Possibly blocking forever due to lack of progress.
- **Contract susceptible to DoS attacks if attacker can cause it to require unbounded gas.**

Vulnerability 1: Unbounded Mass Ops

```
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;
  }
  function openAccount() returns (uint) { … }
}
```

Vulnerability 2: Wallet Griefing

```
for (uint i = 0; i < investors.length; i++) {
  if (investors[i].invested < min_investment) {
    // Refund, and check for failure.
    // Looks benign but locks entire contract
    // if attacked by a griefing wallet.
    if (!(investors[i].addr.send(investors[i].dividendAmount))) {
        throw;
    }
    investors[i] = newInvestor;
  }
}
```

Vulnerability 3: Integer Overflow

```
contract Overflow {
  Payee payees[];

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

uint8

The Vandal Decompiler

nevillegrech.com/2018/madmax.html

Control Flow in EVM Bytecode

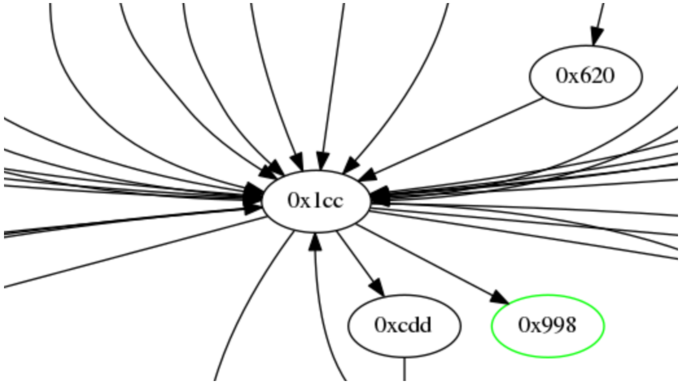
```
         PUSH4 <return>  // return address
         PUSH4 0xFF      // push data
         PUSH4 <foo>     // f      on address
         JUMP            //             'foo'
return:  JUMPDEST
         ...
         ...
foo:     JUMPDEST
         POP             // p     ata
         JUMP            // jumps to 'return'
```

Detect flows of addresses

Decompilation in a Nutshell

1. Basic block boundaries
2. Stack shape and data flow
3. Jump targets
4. Function boundaries
5. Conversion to 3-address IR

Why context sensitivity?



nevillegrech.com/2018/madmax.html

Intermediate Language

to := CONST(c)

where to : Variable , c : Const

JUMPI(cond, label)

where cond : Variable , label : Statement

to := SHA3(index, length)

where index, length, to : Variable

Higher level analyses

nevillegrech.com/2018/madmax.html

Higher Level Analyses

Structured loop reconstruction:

- Induction Variables & Loop Exit Conditions

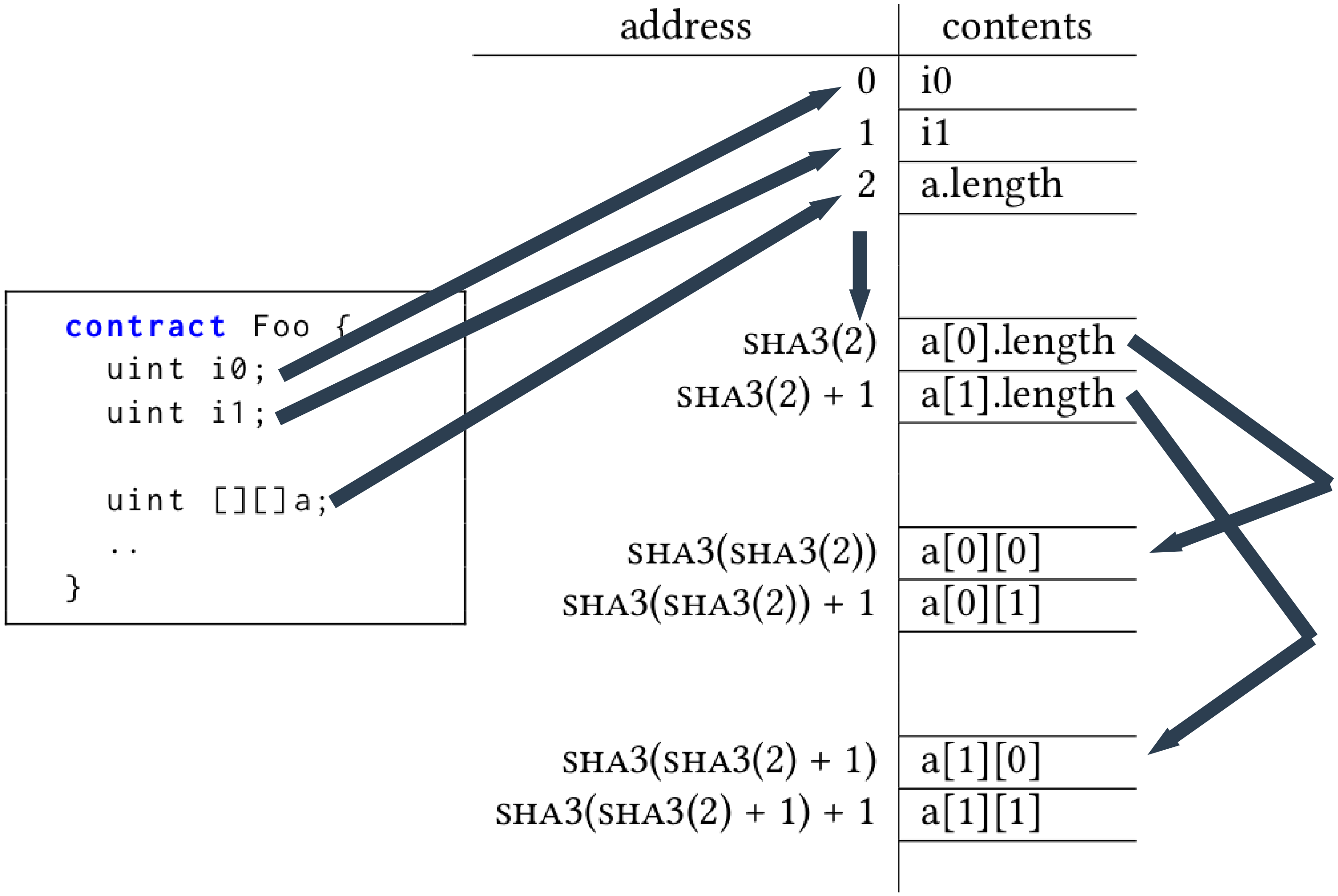
Alias Analyses

High level data structure semantic analysis

Cool concepts such as:

- **`IncreasedStorageOnPublicFunction`**
- **`PossiblyResumableLoop`**

Modeling Storage & Data Structures



nevillegrech.com/2018/madmax.html

Example top-level query

```
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).
```

Experimental Evaluation

nevillegrech.com/2018/madmax.html

Results: Effectiveness

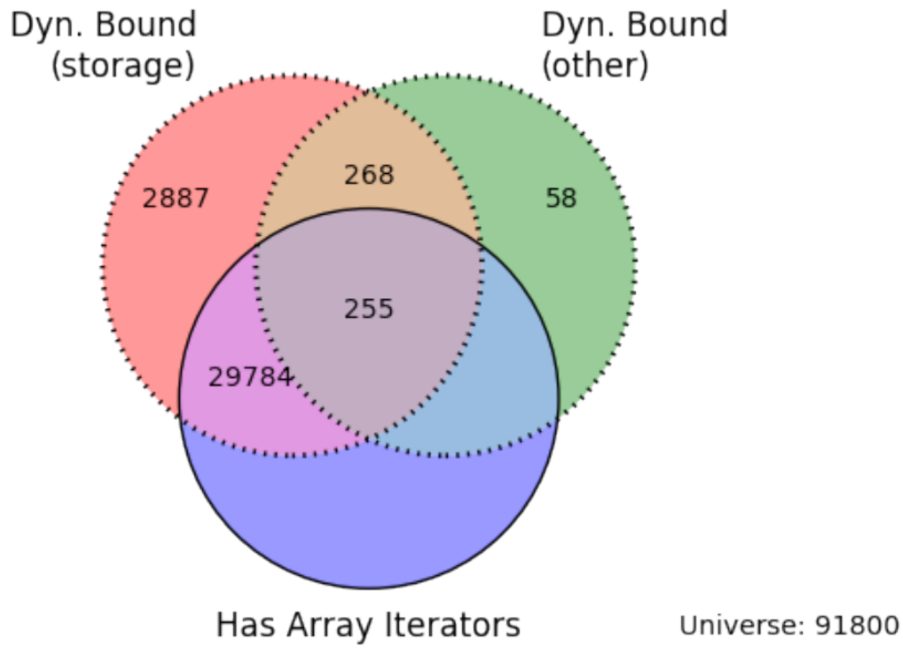
Analysed entire blockchain:

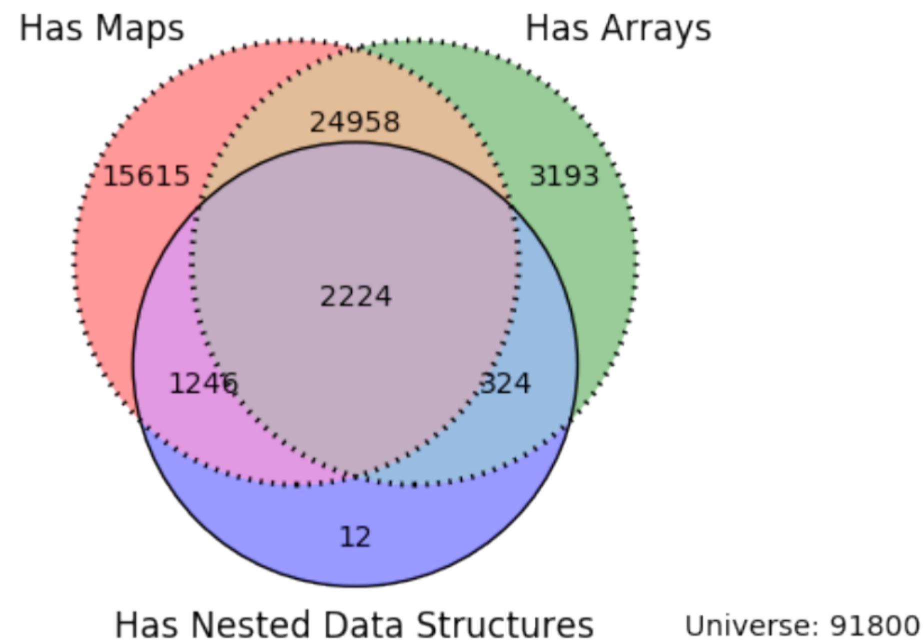
- 6.33M contracts (90k unique) in 10 hours
- 4.1% susceptible to unbounded iteration.
- 0.12% susceptible to wallet griefing.
- 1.2% susceptible to loop overflows.

Combined holding of 7.07 million ETH ~ $5B

81% estimated precision

Insights: Iteration and Data Structures





Reconstructing high level data structure semantics critical for low false positive rate.

Related work

Approach	Works	Soundy	Automated	Bytecode	General
Symbolic Execution	- Oyente by Luu et al. (2016) - Maian by Nikolic et al. (2018) - gasper by Chen et al. (2017) - Grossman et al. (2017)	👎	👍	👍	👎
Formal Verification	- Proofs in Isabelle/HOL by Hirai (2017) & Amani et al. (2018) - Proofs in the K framework by Hildenbrandt et al. (2017) - Formalism of EVM in F* by Bhargavan et al. (2016)	👍	👎	👎	👎
Abstract interpretation on Solidity	- Zeus by Kalra et al. (2018) - FSolidM by Mavridou and Laszka (2018)	👍	👍	👎	👍
Abstract interpretation on EVM bytecode	MadMax (OOPSLA'18) (**Our Approach)**	👍	👍	👍	👍

Conclusions

MadMax, a vulnerability detection tool:

- Scales to the entire Blockchain
- Interesting results, practical impact

Datalog lends itself well to:

- Program analyzers (even flow sensitive ones)
- High level decompilers

Decompilation is a very important step & current work focuses on this

Current work: Fully declarative decompilation

Original Bytecode

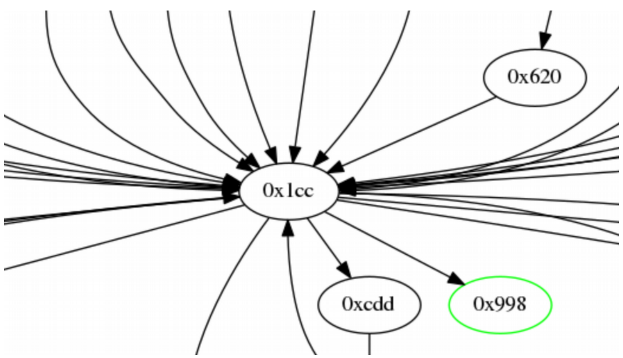
1. Whole contract ctx & flow sensitive analysis

Whole program 3 address IR + CFG

2. Function extraction algorithms

3. Function argument inference with flow sensitive analysis

3 address IR + function bounds + local CFGs



Functional 3-address code