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

NEVILLE GRECH, University of Athens and University of Malta
MICHAEL KONG, The University of Sydney, Australia
ANTON JURISEVIC, The University of Sydney, Australia
LEXI BRENT, The University of Sydney, Australia
BERNHARD SCHOLZ, The University of Sydney, Australia
YANNIS SMARAGDAKIS, University of Athens, Greece

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, a gas limit restricts the execution of an Ethereum smart contract: 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 the 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 a compiler translates into immutable low-level EVM 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 may exploit it 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 executed instruction costs gas, which is traded with the Ether cryptocurrency. Since a user pays gas upfront, a transaction's computation may exceed its allotted amount of gas. As a consequence, the Ethereum Virtual Machine (EVM) raises an out-of-gas exception and aborts the transaction. *A contract that does not correctly handle the possible abortion of a transaction, is at risk for a gas-focused vulnerability.* Typically, a vulnerable smart contract will 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, a contract is susceptible to, effectively, denial-of-service attacks, locking its balance away.

In this work, we present MadMax: 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 producing a high-level program model. MadMax is highly efficient and effective: it analyzes the whole Ethereum blockchain in 10 hours and reports numerous vulnerable contracts holding a total value exceeding $5B, with high precision, as determined from a random sample.

MadMax is unique in the landscape of smart contracts analyzers and verifiers. (Section 7 contains a more detailed treatment of related work.) It is an approach employing cutting-edge *static program analysis* techniques (e.g., data-flow analysis together with context-sensitive flow analysis and memory layout modeling for data structures), whereas past analyzers have primarily focused on symbolic execution or full-fledged verification for functional correctness. As MadMax demonstrates, static program analysis offers a unique combination of advantages: very high scalability, universal applicability, and high coverage of potential vulnerabilities.

We speculate that past approaches have not employed static analysis techniques due to three main reasons: a) the belief that the thoroughness of static analysis is unnecessary for smart contracts since they are small in size; b) the possibility that static analysis, although thorough, can yield a high number of false positives—full-fledged, less automated verification techniques may be necessary; and c) the difficulty of applying static analysis techniques uniformly, at a low level: decompiling the low-level EVM bytecode into a manageable representation is a non-trivial challenge.

MadMax addresses or disproves these objections. It provides an effective decompilation substrate for analyzing low-level EVM bytecode. MadMax exhibits high precision, due to the sophisticated modeling of the gas-focused concepts it examines. Finally, our study of the Ethereum blockchain (and the subsequent application of MadMax to it) reveals that smart contracts can significantly benefit from static analysis. Figure 1 gives an early indication, by plotting smart contract size against the Ether held. We can see that relatively complex contracts (measured in the number of basic blocks) contain most of the Ether. Hence, the potential risk compounds for sophisticated smart contracts because complex contracts are harder to get right. This observation strongly supports the use of static program analysis, which scales well to relatively complex programs.

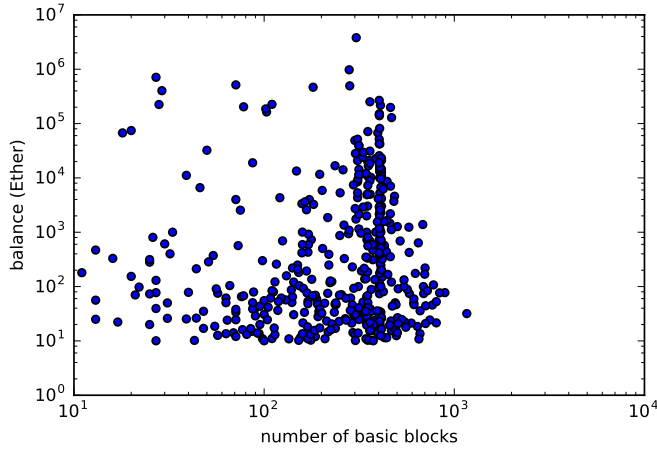The main contributions of our work are:

Fig. 1. Complexity (as the number of basic blocks) vs. balance held for all smart contracts as of April 2018.

- **A decompiler from EVM bytecode to structured low-level IR:** We propose the use of static program analysis directly on the EVM bytecode. Analyzing EVM bytecode is challenging due to the stack-based low-level nature of the EVM with minimal control-flow structures.
- **The identification of gas-focused vulnerabilities:** The semantics of limited, gas-based execution on top of smart contracts handling monetary transactions introduces a new class of vulnerabilities that does not occur in other programming language paradigms. We identify out-of-gas vulnerabilities thoroughly and explain their essence.
- **Abstractions for high-level data-structures and program constructs:** We construct high-level abstractions for EVM bytecode for bridging the gap between the low-level EVM and the high-level vulnerabilities. We express analysis concepts that include safely resumable loops, data structures whose size increases in repeat invocations of public functions, and recognition of nested dynamic structures in low-level memory.
- **Validation:** We validate the approach for *all 6.3 million contracts* deployed on the entire blockchain. To our knowledge, no other work in the smart contract security literature has performed program analysis on such a number of contracts. Our analysis does not require source code to run, nor external input, and at the same time is highly scalable. The analysis reports vulnerabilities for contracts holding a total value of over $5B. Even though it is uncertain whether most vulnerabilities are real and how easily exploitable they might be, manual inspection of a small sample reveals over 80% precision and the existence of specific issues, which we detail.

2 BACKGROUND

Next, we provide a concise background on the setting of our work including blockchain platforms, Ethereum smart contracts, and the language abstractions behind them.

2.1 Blockchain Platforms, Ethereum, and Smart Contracts

A blockchain is a shared, transparent distributed ledger of transactions, 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 special software 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 arises, 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, since 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 multi-party interactions with trust arising from the majority of peers, instead of being given by default to an authority. This property has created excitement and led to well-founded assertions that "*Blockchain is a foundational technology: It has the potential to create new foundations for our economic and social systems*" [Iansiti and Lakhani 2017].

The original blockchain, at least in its popular form, is due to the Bitcoin platform [Nakamoto 2009]. Bitcoin is explicitly a special-purpose cryptocurrency platform. Therefore, the data registered on the Bitcoin ledger can be seen as (transaction and balance) amounts. The blockchain formulation we are interested in is the one popularized by the Ethereum platform [Buterin 2013; Wood 2014]: registered accounts are not mere balances, but may contain smart contracts, i.e., code that can perform arbitrary computations, enabling the encoding of complex logic. Virtually any complex interaction can be captured, with its logic permanently and transparently stored on the blockchain, responding to communication from other accounts, dispensing or accepting funds, etc. The possibilities for such logic are endless: it can encode a payoff schedule, investment assumptions, interest policy, conditional trading directives, payment-upon-receipt agreements, pricing dependent on geographic or other environmental input, and a lot more. Gaming, music distribution, remote purchases, and many other business interactions can utilize automatically-enforced smart contracts for secure and verified transactions, without a need for intermediaries or third-party trust.

2.2 Smart Contract Programming: Solidity and the EVM

Ethereum smart contract programming is most commonly done in the Solidity language [Various 2018c]. Solidity is a JavaScript-like scripting language, enhanced with static types, contracts as a class-like encapsulation construct, contract inheritance, and numerous other features.

Although Solidity is commonly used, it is far from the only language for writing smart contracts (others are Serpent, Viper, and LLL [Various 2018a,b,d]). Furthermore, its 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 including 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, in the form of persistent data structures, or in contract-local memory.

The EVM supports a segregated, near-infinite (256-bit) memory space per contract. Addresses are cryptographic hashes so that a dynamic structure's location in the virtual memory space is merely the hash of the structure's identifier. This model is an unconventional memory layout and is not straightforward to discern in low-level code. The EVM bytecode language is type-less and unstructured, with several low-level elements, as will be discussed in Section 4.1.

2.3 Analysis Level

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) and, therefore, not commonly encountered in past work on contract analysis.

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. It is estimated that high-level source code of EVM contracts is only available for a tiny fraction (just 0.34%) of contracts on etherscan.io[etherscan.io [n. d.]]. Various high-level languages that target the EVM are in use and, additionally, Solidity contracts can be interleaved with inline assembly, which is similar in structure to EVM bytecode.

Furthermore, the impedance mismatch between a high-level language and the EVM bytecode is often a source of confusion and errors. For instance, consider the code pattern below:

```
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. (This code was behind a vulnerability [Atzei et al. 2016] in the GovernMental [Various [n. d.]a] smart contract, for example.) Other similar patterns abound—we will also see an illustration in a common gas-focused vulnerability in Section 3.3.

Therefore, a reliable analysis needs to capture the full semantics of Solidity, which is often far from trivial. An analysis at the EVM bytecode level eschews this complexity, instead of dealing only with features that offer uniform power, for all contract source languages.

Even if Solidity code is available, an analysis at the abstract syntax tree (AST) or text level will need to handle several code complexities and indirections, such as performing intermediate assignments before using a value, or calling a function to perform an effect on storage, instead of doing it locally. This limits the power of AST-based or text-matching (grep-like) techniques for vulnerability analysis. Instead, at the EVM bytecode level, the input is normalized, with all control flow being explicit, uniform, and simplified.

3 EVM GAS-FOCUSED VULNERABILITIES

In this section, we identify some of the most common patterns of gas-focused vulnerabilities and illustrate them with examples. We employ Solidity for demonstration purposes, even though our entire analysis work is at the EVM bytecode level.

The mechanism underpinning Ethereum contracts is designed to incentivize users to minimize the number of instructions executed by making them pay up front for the gas required to execute a single transaction. Running out of gas is common, but, in most cases, this is not catastrophic: the transaction is reverted and re-run with a higher gas budget by the end user. Unfortunately, this is not always an option. Many times the amount of gas budgeted is hard-coded in other contracts that have already been deployed, so it cannot be increased. Furthermore, Solidity's send or transfer implementation, which can call other smart contracts, can only send 2300 units of gas. Finally, the Ethereum network has a block gas limit that cannot be surpassed. Therefore, contracts may reach a state (e.g., the size of a data structure in storage can grow a lot) such that they will never have enough gas to run their code.

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 the loop body too many times exceeding the block gas limit (or becomes 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 below:

```
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. Eventually, the function may be impossible to execute without raising an out-of-gas exception.

Ethereum programming safety recommendations [Various [n. d.]b] 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). This practice is typically phrased in terms of payments ("favor pull over push payments") although it applies more generally. (As we shall see, a large number of existing contracts, holding very substantial sums, violate this practice.)

A further 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" [Various [n. d.]b].

The above NaiveBank function can be re-written so that it can resume if it did not manage to complete all its operations as follows:

```
  Account accounts[];
  uint nextAccount;

  function applyInterest() returns (uint) {
    for (uint i = nextAccount; i < accounts.length && msg.gas > 100000; i++) {
      // apply 5 percent interest
      accounts[i].balance = accounts[i].balance * 105 / 100;
    }
    nextAccount = i < accounts.length ? i : 0;

    return accounts.length;

  }
```

This pattern is, as we determine, rarely used correctly in existing contracts, demonstrating the community's unawareness of the dangers inherent in certain smart contract designs.

3.2 Non-Isolated External Calls (Wallet Griefing)

In addition to running out of gas because of unbounded, externally-controlled data structures, a contract may run into trouble because of invoking external functionality that may itself throw an out-of-gas exception. This is not a realistic threat in a direct setting: an external call to an unknown party is by definition untrusted, and therefore the contract programmer is highly likely to have considered malicious behavior. Note that Solidity primitives are also designed to encourage such defensive design. However, the vulnerability is realistic when combined with several other complicating factors: implicit code invocation at an Ether transfer, handling multiple clients without isolation, and standard practices for aborting on a send failure.

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. Therefore, sending Ether may fail, as it invokes arbitrary, possibly untrusted, code. The failure is usually, but does not have to be, an out-of-gas exception.

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 2,300 given to the callee. This is typically 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 also used in recommended Ethereum security code patterns [Various [n. d.]b] is to use 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.[1] Naïve error handling, i.e., just aborting the transaction, is no longer sufficient. We can see the issue in example code for a vulnerability appealingly termed *wallet griefing* [Vessenes 2016].[2] Consider a simple fragment (adapted from [Vessenes 2016]) of code that tries to refund investments below a minimum amount:

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

---

[1]The pattern of doing external calls in a loop is also avoided by the "favor pull over push payments" recommendation, which is, however, still, just a recommendation, and has only recently reached broad awareness.

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

The problem is that the send command will also result in the callback function of the selected investor being executed. All it takes for the contract to be vulnerable is for an attacker to make themselves an investor below the minimum threshold, and then provide a callback function that runs out of gas.

Importantly, it is not enough for the above loop to be resumable, as in Section 3.1. Picking up from the same point will not lead to progress (as it would have if the contract had merely run out of gas): the failed send will re-fail if repeated. More advanced handling of failure-to-send errors is required, so that, e.g., future executions of the contract will be advised that the earlier Ether transfer failed (and will not re-attempt it blindly).

3.3   Integer Overflows

A programming error that commonly expresses itself as a gas-focused vulnerability results from the Solidity type inference approach and its resulting possible integer overflows. This is a separate pattern from the general attack of Section 3.1, since the iteration is not merely unbounded but literally non-terminating, due to an overflow condition producing unexpected values.

Consider the following contract that exhibits an insidious bug:

```
contract Overflow {
  Payee payees[];

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

The use of var induces the problem in the type inference. The inferred type of variable i is uint8 (i.e., a byte), since 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.

Beyond unexpected type inference, there are indeed more possibilities for overflow in realistic contracts. As we discuss in our evaluation, integer overflows may also tend to creep in when the contract designer is trying to optimize storage requirements.

3.4   Possible Attacks and Incentives

The most direct exploitation of the vulnerabilities identified above would result in locking down a contract, rendering it unusable. The attacker typically needs to expend funds to exploit a vulnerability—e.g., consume Ether to populate a dynamic data structure. In the cryptocurrencies ecosystem, some incentives for performing such an attack include:

- bringing down the price of ETH;
- amassing fame in the relevant subcommunity;
- destroying competitors' contracts;
- blackmailing of contract holders (of the contract under attack, or any other).

4   ANALYZING 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 non-trivial, since EVM

bytecode is low-level: much closer to machine-specific assembly than to structured IRs (e.g., Java bytecode or .NET IL). We explain the challenges and decompilation techniques in the next sections.

Our decompiler, code named Vandal [Various 2018], has been released as a freely-available, self-standing tool.

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 highlight this issue, compare the EVM bytecode language to the best-known bytecode: Java (JVM) bytecode—a much higher-level IR. The design differences include:

- Unlike JVM bytecode, EVM does not have the notion of structs or objects, nor does it have a concept of methods.
- Java bytecode is a typed bytecode, while EVM bytecode is not.
- In JVM bytecode, 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 structure exists, which means that standard control-flow structures are more accessible to infer from JVM bytecode than from EVM.
- All control-flow edges (i.e., jumps) 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 can be resolved, 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 intra-contract 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.

These challenges have often prevented past work from attempting a static analysis directly on the EVM bytecode. Instead, the literature is more abundant in semi-dynamic techniques, such as symbolic execution, as we discuss in Section 7.

4.2 Vandal Decompiler Analysis

The Vandal decompiler [Various 2018] 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.

Because of the complexities we saw in Section 4.1, producing a structured intermediate representation for an EVM bytecode program is not as simple as for other stack-machine IRs (e.g., the conversion of JVM bytecode into the Jimple intermediate language in the Soot framework [Vallée-Rai et al. 1999; Vallée-Rai et al. 2000]).

Our earlier discussion directly suggests the challenges:

- Detecting basic blocks is trivial (because of explicit jump labels), but identifying their connectivity to form a control-flow graph (i.e., the possible targets of every jump) requires a

value-flow analysis. The same is true for detecting possible values (or types) of arguments for every operation, that is, the shape of the stack (i.e., its size and static types at every position).

• Detecting function boundaries requires a high-fidelity analysis. Common patterns in the compilation of contract code exacerbate this, e.g., in the Solidity compiler, since most smart contracts are written in Solidity. The Solidity compiler translates local contract calls into jumps to a *dispatcher* routine, which then redirects the call to the initial basic block of the target function.

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 [1999]. The only differences in our setting are somewhat superficial: instead of a "call" instruction, we have jumps, which could well be encoding non-call control-flow transfer (i.e., inside a single function). However, it is possible to recognize, with only local code inspection, jumps that push a return address (i.e., forward calls with a continuation) and distinguish them from local jumps and non-local returns.

The setting of CPS input and needing to detect value and control flow is precisely that of *control-flow analysis (CFA)* [Shivers 1991, 2004]. 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.

This is a good fit for the need of our analysis to detect calls that go through the dispatcher code emitted by the Solidity compiler. Beyond forming the CFG, we want to distinguish flows that go through the dispatcher, where a single caller (i.e., block that jumps to the dispatcher) in reality always reaches a single target (i.e., block that the dispatcher jumps to). Consider the control-flow graph fragment shown in Figure 2 (automatically produced by Vandal using a contract scraped from the blockchain).

As can be seen in the figure, the dispatcher node (0x1cc) has a high in-degree and out-degree. However, flow through the dispatcher may disguise a more informative calling pattern. For instance, block 0x620 may always be calling block 0x998 and not any of the other blocks that the dispatcher calls out to. A 1-CFA analysis is likely enough to track the flow through the dispatcher when the latter is used for straightforward transfer of control between local functions of the same contract: the dispatcher code is analyzed separately for each of its predecessors.

Fig. 2. Part of control-flow graph produced for the actual contract. Dispatcher block shown in the middle.

Vandal, therefore, adopts the standard form of a control-flow analysis [Shivers 2004], formulated as an abstract-interpretation. There are a few realistic design choices that we employ:

• We implement context sensitivity using the cloning of basic blocks. Cloning proceeds breadth-first: initially at depth 1 (essentially capturing a 1-CFA analysis), then at depth 2, 3, etc., with no *a priori* bound. The process terminates either a) when recursion is detected; or b) when there is no cloning to be performed at the given depth; or c) when a global timeout (e.g., 20sec in our current implementation) for the analysis is reached. This means that the context
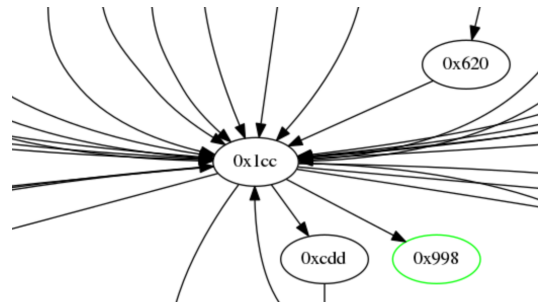
sensitivity of our analysis is in principle unbounded. However, in practice it rarely goes beyond a depth of 2 due to the global timeout. Reaching the timeout is an indication of the analysis being unscalable: the practice of cloning basic blocks (i.e., inlining called functions, with no depth limit) has led to an explosion of the analysis model, up to much larger-than-expected levels. Such an explosion of analysis information typically would also indicate (at least partial) imprecision in analysis results.

- We short-circuit value sets to a ⊤ ("top") value, if they grow larger than a pre-set number of elements, m, (e.g., 5 in our current setting). In terms of the abstract interpretation formulation of the analysis, this is a common-practice restriction of the abstraction lattice, to be a powerset-lattice (i.e., with sets of values as points) for sets with size up to m.

4.3 Schema of Decompiler Output

Vandal produces 3-address code using the schema listed in Figure 3. 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, i.e., it denotes any variable, which is ignored.

The schema captures all elements of EVM bytecode, slightly abstracted. The form is that of a standard, structured intermediate language. As can be seen, JUMPI instructions have statements, and not arbitrary values, as targets. All binary operations are treated equivalently, since we currently do not attempt to analyze arithmetic. We do not include unary operations or direct assignment between variables in the figure although we do so in the implementation, since these can be treated as special cases of binary operations. RUNTIMEKNOWABLE gives a uniform treatment of instructions that return the cost of gas, transaction id, code size, caller, and other run-time quantities.

5 ANALYSIS OF GAS VULNERABILITIES

The MadMax pipeline, from the Vandal decompiler to the final analysis output, is depicted in Figure 4. The main MadMax analysis operates on the output of Vandal using logic-based specifications. The analysis is implemented in the Datalog language: a logic-based language, equivalent to first-order logic with recursion [Immerman 1999]. The Datalog implementation treats the relations of Figure 3 as its input schema. We also use Datalog in the paper, as a means for precise specification of the analysis. Although the form of the rules presented here is simplified relative to the full implementation, the essence is kept unchanged.

The MadMax analysis consists of several analysis layers that progressively infer higher-level concepts about the analyzed smart contract. Starting from the 3-address-code representation, 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 re-entry, nested arrays, etc. This step is non-trivial, since it needs to model the EVM dynamic data representation. 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

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)]   where $s : S, to : V, c : C$

load from storage
s:[to := SLOAD($index$)]   where $s : S, index : V, to : V$

store to storage
s:[SSTORE($from, index$)]   where $s : S, index : V, from : V$

load from (volatile) memory
s:[to := MLOAD($index$)]   where $s : S, index : V, to : V$

store to (volatile) memory
s:[MSTORE($from, index$)]   where $s : S, index : V, from : V$

conditional jump
s:[JUMPI($cond, label$)]   where $s : S, cond : V, label : S$

conditional throw
s:[THROWI($cond$)]   where $s : S, cond : V$

keccak 256 hash
s:[to := SHA3($index, length$)]   where $s : S, index : V, length : V, to : V$

call external contract
s:[to := CALL($addr, gas$ ...)]   where $s : S, addr : V, gas : V, to : V$

get remaining gas
s:[to := GAS()]   where $to : V$

get run-time value (e.g. current block size)
s:[to := RUNTIMEKNOWABLE()]   where $to : V$

CAST integer to a number of bits
s:[to := CASTN($from$)]   where $to : V, from : V, n : \mathbb{N}$

binary operator e.g. ϕ, ADD, AND, etc.
s:[to := BinOp(a, b)]   where $s : S, a : V, b : V, to : V$

Fig. 3. Our domains and decompiler output primitives (i.e., input relations for main analysis) for EVM 3-address code.
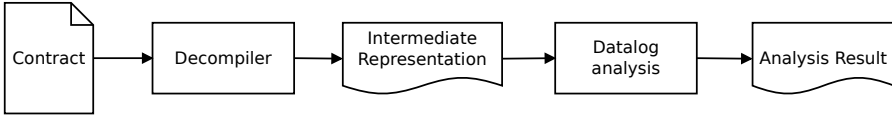
Fig. 4. The MadMax analysis pipeline

	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

Fig. 5. Extra domains, input, and output schema for baseline loop and data flow analyses.

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 5. 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 frameworks for Java bytecode, such as JChord [Naik 2011; Naik et al. 2009] or Doop [Bravenboer and Smaragdakis 2009].

The first three computed relations in Figure 5 (INLOOP, INDUCTIONVAR, and LOOPEXITCOND) encode useful concepts in structured loops. Note that loops in low-level programs do not have to be structured, e.g., there may not be a loop head that dominates all loop statements. However, Solidity and other EVM languages produce structured loops as part of their compilation process. The loop analysis finds induction variables, i.e. 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, i.e., it ignores storage and 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, MEMCONTENTS) is best-effort, i.e., 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, run-time operations, memory load and stores, etc., without the analysis computing this. Second, not all inferences are guaranteed to hold. E.g., an inference that is known to hold in one control-flow path but not in another will be optimistically propagated when paths are merged.

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 := RUNTIMEKNOWABLE()],
  LoopBoundBy(loop, dynVar).

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

Fig. 6. Inferencing bound loops and resumable loops

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. To quote Flanagan et al. [2002]: *Insisting that the checker meet either ideal [soundness or completeness] is mistaken on engineering grounds: if the checker finds enough errors to repay the cost of running it and studying its output, then the checker will be cost-effective, and a success. To achieve a cost-effective tool requires making good engineering trade-offs between a variety of factors, including: missed errors (unsoundness), spurious warnings (incompleteness), annotation overhead, and performance.*
However, compared to past techniques that do not employ static analysis, MadMax can be argued to be *soundy* [Livshits et al. 2014]: it makes an effort to achieve soundness (i.e., exhaustive modeling of *all* program behaviors) up to features that would make its modeling so imprecise as to be unscalable.

- 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. For example, in Section 5.2 we define a relation RelatedKeys, which is transitively closed with respect to VarAlias: variables used as data structure keys that are related, make their aliases also related. For purposes of simplicity, 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 6. If both an induction variable i and a non-induction 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 [Various [n. d.]b], 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, e.g., gas check, store, load of induction variable. 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.

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]

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

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

Fig. 7. Storage structure and contents (right) for given contract (left). SHA3 is the KECCAK256 hash function.

5.2 Memory Layout and Analysis of Data Structures

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. After all, gas limits are always finite numbers, at most up to a constant (the block limit). Would not any unbound iteration be in principle vulnerable, especially since as fully resumable loops are rare? Unfortunately, a precise analysis requires more sophistication. We find experimentally that around half of the currently deployed contracts have dynamically bound loops—it would be entirely unrealistic to expect that half of smart contracts currently deployed are vulnerable. This should not be a surprising insight—a program's behavior largely depends on its input and persistent state. Many of these dynamically-bound loops are in fact iterating over data structures, as we shall see in Section 6.3.

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, i.e., maps. Solidity offers fixed-sized arrays or structs as well. Throughout the development of our high-level analyses, we found that modeling dynamically-sized data structures is essential for precisely detecting gas-focused vulnerabilities. For example, an application that iterates over a data structure in storage is not at risk of most vulnerabilities if the data structure is of a fixed size.

Although both arrays and maps can be dynamically resized, no mechanism exists for iterating over maps. Instead, contract implementers need to keep track of entries on both arrays and maps to be able to both index by key or iterate through entries. Furthermore, all of the gas-focused vulnerabilities that involve data structures also involve unbounded loops making arrays the prime data structure to model.

The Ethereum memory layout is highly unconventional from a traditional programming languages standpoint, although perfectly reasonable if one considers the specifics of the execution

KEYTOOFFSET(*keyVar, structureOffsetVar, keySize*) ←
   shaStmt:[*structureOffsetVar* := SHA3(*start, keySizeVar*)],
   MEMCONTENTS(*shaStmt, start, keyVar*),
   HASCONSTANTVALUE(*keySizeVar, keySize*),
   VARINDEXESSTORAGE(*, structureOffsetVar*).

RELATEDKEYS(*keyVar, keyVar', keySize*),
KEYTOOFFSET(*keyVar, structureOffsetVar', keySize*) ←
   KEYTOOFFSET(*keyVar, structureOffsetVar, ** *),
   FLOWS(*structureOffsetVar, keyVar'*),
   KEYTOOFFSET(*keyVar', structureOffsetVar', keySize*).

ARRAYIDTOSTORAGEINDEX(*arrayId, storeOffsetVar*) ←
   KEYTOOFFSET(*keyVar, storeOffsetVar, 32*),
   HASCONSTANTVALUE(*keyVar, arrayId*).

ARRAYSIZEVARIABLE(*sizeVar, arrayId, keyVar*) ←
   RELATEDKEYS(*keyVar, keyVar', 32*),
   [*sizeVar* := SLOAD(*keyVar', ** *)],
   HASCONSTANTVALUE(*keyVar, arrayId*).

VARINDEXESSTORAGE(*stmt, var*) ←   *stmt*: [SSTORE(*var, ** *)].
VARINDEXESSTORAGE(*stmt, var*) ←   *stmt*: [* := SLOAD(*var, ** *)].

Fig. 8. Datalog rules for recursively identifying dynamic data structures.

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 7 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 calculation 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, are 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.

An analysis for identifying dynamic data structures in smart contracts is shown in Figure 8. Relation KEYTOOFFSET links variables that are assigned with a data structure's key (array or map key) with a variable that contains said structure's offset in storage. Variable *keySize* represents the size of the key, i.e., the number of bytes hashed to produce the offset in storage. This is a constant value in the bytecode, as provided in the rule. In the case of arrays the key size is 32.

RELATEDKEYS relates data structure keys to each other if one is contained in the other or if there is aliasing. Effectively, if *keyVar* was used to compute *keyVar'*, which is also used as an offset, the latter is likely to be a substructure of the former.

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, **).

Fig. 9. Datalog rules for identifying storage requirements increase in public functions.

ARRAYIDTOSTORAGEINDEX maps variables that represent offsets in storage to a unique array identity. For instance, array a in Figure 7 has an identity of '2'. This is static knowledge, encoded in the program itself: every variable has a static memory offset. It is worth noting that the rules are general enough to map any substructure, even if, for instance, an array is inside a map or vice versa. All substructures are mapped back to their top-level structure's identity. Note that arrays are identified using a single 32 byte word, whereas map items are identified by two 32 byte words (i.e. 64 bytes).

The above relations, together with VARALIAS (for local variable aliasing—see Section 5.1) form the MadMax static memory model. The model is not fully general. Notably, alias analysis is only local—heap memory is modeled fully only for arrays and maps, not for arbitrary low-level address manipulation. Still, arrays and maps are the native Solidity data structures, and all others are typically built out of them. The model captures the information necessary for identifying substructures and the variables that may refer to them.

Finally, to statically reason about storage requirements, the analysis needs to find which variables represent array sizes, as captured by relation ARRAYSIZEVARIABLE. Note how the definition uses RELATEDKEYS, so that the sizes of all sub-arrays also get associated with the containing array, to capture all possible size changes conservatively.

Based on the above relations that model the memory layout, we define key concepts for gas-focused analyses, as shown in Figure 9. An important concept is INCREASEDSTORAGEONPUBLIC-FUNCTION. 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 is bound by an array's size. This uses our previously defined query on loops DYNAMICBOUND.

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 10 shows the final output relations of the MadMax analysis in inlined-to-single-rule and in a slightly simplified 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 throw need to be in the same loop, which also has an induction variable that affects the address loaded.

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*) ←
  DYNAMICALLYBOUNDLOOP(*loop*),
  [*to* := CASTN(*from, n*)], $n \leq 16$,
  INDUCTIONVAR(*to, loop*),
  INDUCTIONVAR(*from, loop*),
  FLOWS(*to, condVar*),
  LOOPEXITCOND(*condVar, loop*).

Fig. 10. Top level query for unbounded mass operations, wallet griefing and overflow vulnerabilities.

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, i.e., the number of iterations is determined by some run-time value.

The common theme behind our analyses for these vulnerabilities is that we only look for them in the presence of loops. There is further commonality between UNBOUNDEDMASSOP and the WALLETGRIEFING analysis: to exploit any of these, the attacker must be able to append elements to the array that is iterated upon. An important difference, however, is that having resumable loops that keep track of gas will not help against wallet griefing, as the attack only depends upon Ether transfer to an uncooperative fallback function, which can be set up as part of the attack.

6   EVALUATION

In this section, we present the results of our evaluation of MadMax. MadMax is implemented in Python for the Vandal decompiler and scaffolding, and in Datalog for the main analysis. The decompiler produces 3-address relations in a normalized form that are loaded into a Datalog-based database. We use Soufflé [Jordan et al. 2016] as our Datalog engine. Soufflé compiles its Datalog

input program into a C++ application, with specialized data structures for performing relational operations. In addition, we have implemented blockchain scrapers to get contracts and data off the Ethereum blockchain. Our experimental setup consists of all programs available on the Ethereum blockchain. This makes up the universe set of "contracts in the wild".

We ran MadMax on an idle machine with an Intel Xeon E5-2687W v4 3.00GHz and 512GB of RAM. Due to time constraints, we set a cutoff of 20s for decompilation. Any contracts that take longer to decompile are considered to timeout.

There are several research questions that our experiments intend to answer:

RQ1. Is MadMax an effective static analysis?
  RQ1.A. Is the analysis relevant? What percentage of contracts are vulnerable and how much currency do these hold?
  RQ1.B. Is the overall analysis precise, i.e., does it have a low false positive rate?
RQ2. Is MadMax an efficient analysis?
RQ3. Are the individual MadMax components and design decisions justified?
  RQ3.A. Is the Vandal decompilation approach effective (in terms not captured by the earlier overall effectiveness of MadMax)?
  RQ3.B. Do the insights from actual contracts vindicate the design choices for the analysis and the patterns it captures?

6.1 RQ1: Effectiveness of MadMax

We evaluate the relevance of the analysis and vulnerabilities by examining the number of contracts that are flagged as vulnerable and the amount of currency (ETH) these contracts hold. Naturally, this metric has to be qualified by a false positive rate (RQ1.B). This is a question that we can only answer approximately: we take a sample of contracts and manually inspect them to see whether the contracts are really vulnerable, and comment on the insights acquired by the inspection.

The contracts that are flagged for vulnerabilities, combined, contain 7.07 million ETH, or roughly $5.7 billion.[3] In total there are 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, since 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%. By manually inspecting these contracts we have gained important insights about the effectiveness of MadMax. We present a subset of them next. We refer to the manually inspected contracts as Contract #1 to Contract #13.[4] The subset of contracts selected for manual inspection include casino and gambling contracts, bidding contracts, ICOs, multi-sig wallets and even a "marriage contract".

Contract #1 (a betting contract) is flagged for wallet griefing, unbounded loops, and loop overflow. The wallet griefing case is a false positive as we could not establish it through manual inspection. We noticed that it is likely that the inferred control-flow graph has imprecision due to a simple function being used in many locations. The contract is also flagged for unbounded iteration and

---

[3]The average price of ETH/USD in 2018 as of Apr 13th 2018 is $807.54, all balances as of Apr 9th.
[4]To avoid misuse we will not publish exact details on these contracts until we have in place a responsible disclosure protocol.

loop overflow, and indeed it is vulnerable to these. In particular, loop overflow will certainly happen if the contract owner tries to withdraw and there are more than 256 bets for the same event. Interestingly, the developer is aware of possible loop overflows and, for instance, acknowledges checks for such an eventuality in other parts of the contract:

```
if (balanceOf[_to] + stake < balanceOf[_to]) throw; // Check for overflows
```

Contract #2 is flagged for unbounded loop iteration, but since there is an additional check that breaks the loop when the size of a data structure is too large, the warning is actually a false positive.

Contract #3, flagged for wallet griefing and unbounded mass iteration (both true), does not use a standard `for` loop for iterations, which is not a challenge for our analysis. We have simplified/obfuscated some details in this contract for security and readability reasons, as follows:

```
contract Contract {
  XYZ[] public xyzs; ...
  struct XYZ {
    uint256 value;
    address owner;
  } ...
  uint256 public index;
  function work() public payable { ...
    while (index < xyzs.length && ...) {
      XYZ storage xyz = xyzs[index];
      ...
      xyz.owner.transfer(...);
      ...
      if (...) { ... index++; ... }
    } ...
    xyzs.push(XYZ({
      value: ...,
      owner: msg.sender
    }));
  }
}
```

We can see that the wallet griefing attack for this contract is simple to execute and involves calling the `work` function with the right amount of ether, i.e., generally a small sum that satisfies some of the requirements for the while loop to start. The attacker has to construct a fall-back function that throws an error. After the loop ends, the attacker's address is stored and the next time `work` is called, the contract would throw exceptions in the loop once it iterates over the attacker's details. This will happen every single time.

Contract #4 increases the size of the data structure `m_pendingIndex` by directly manipulating the `length` field of the array. Our bytecode-level analysis has no problem capturing this behavior:

```
bytes32[] m_pendingIndex; ...
function confirmAndCheck(bytes32 _operation) returns (bool) { ...
  var pending = m_pending[_operation];
  // if we're not yet working on this operation,
  // switch over and reset the confirmation status.
  if (pending.yetNeeded == 0) { ...
    pending.index = m_pendingIndex.length++;
    m_pendingIndex[pending.index] = _operation;
  }
```

```
}
```

For lack of space, we summarize the findings of the contracts. Contract #5 uses a multi-dimensional (stacked) data structure (i.e., map of structs that contain arrays), which increases in size. The analysis identifies this behavior correctly. Contract #7 is falsely flagged for unbounded iteration. Contract #9 is flagged for loop overflows, and despite having the implementation of operations specifically designed to not overflow, the implementer overlooks one of the loops. Similarly, Contract #11 uses a library called Safemath, but it is not used everywhere, resulting in a true warning. Contract #12 tries to use assertions to avoid overflows but some corner cases are missed. Contract #13 uses a `while` loop on a decreasing quantity, subtracting a dynamically-determined value. This contract is vulnerable to underflows for some inputs.

6.2 RQ2: Efficiency of MadMax

MadMax is able to analyze (modulo timeouts) the 91.8k contracts on the blockchain in around 10 hours on our hardware configuration, running 45 concurrent processes. Each contract takes an average of 5.6s for decompilation and 0.3s for the client analyses, making this a very scalable analysis.

The Vandal decompiler timed out in 8.2% of the contracts (with a 20s timeout). By inspection, we see that these contracts are of larger-than-average size, but they are not the largest nor exhibit consistent patterns different from the rest of the contracts. In principle, timing out should not prevent getting partial decompilation results for these programs: Vandal could report results produced for up to the previous inlining depth explored. However, the current Vandal implementation has engineering limitations, making the graceful handling of timeout not always possible. We believe that this restriction can be addressed in future work.

6.3 RQ3: Analysis of Components and Design Decisions

RQ3 is evaluated quantitatively by interpreting metrics derived from all Ethereum contracts.

6.3.1 RQ3.A. Vandal decompiler effectiveness.
Vandal decompilation captures most program behaviors. This is certainly suggested by the earlier effectiveness results of the overall MadMax pipeline. However, we have also empirically established that in 75% of programs, all jump targets, under all call-site contexts are resolved. Note that if a single jump in the program is missing one of its two targets—e.g., because of behavior too complex for the analysis to resolve, the entire smart contract is considered unresolved. The contract becomes part of the 25%.

### 6.3.2 RQ3.B. Analysis Design Decisions vs. Contract Metrics.

Dynamically bound loops. Figure 11a shows that half of the contracts have dynamically bounded loops; many of these are in fact iterating over data structures. This fact highlights the importance of fully analyzing data structures to reason about the resource behavior of smart contracts.

Loops bounded by gas. Although checking and budgeting loops by gas at run-time is sometimes recommended as a way to avoid gas-focused vulnerabilities, this pattern has not been taken on board by smart contract programmers as shown in Figure 11b.

Arrays used for iteration. Iterating loops over arrays is a surprisingly common pattern in smart contracts, as we can see in Figure 12a. These arrays typically represent user accounts, bids, game rounds, etc., and can sometimes be manipulated by simply calling public functions.
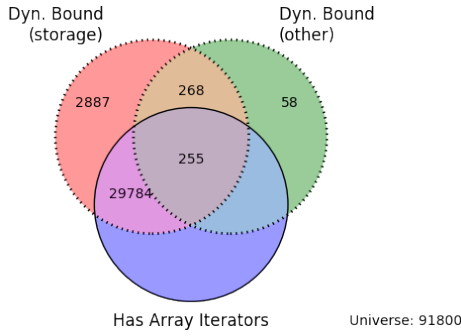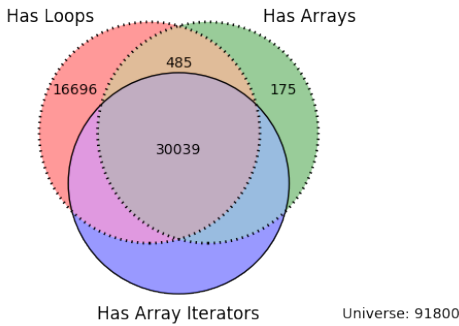
(a) Prevalence of dynamically bound loops (categorized further into storage and other) and contracts that have array iterators. Most dynamically bound loops are actually iterating on arrays.
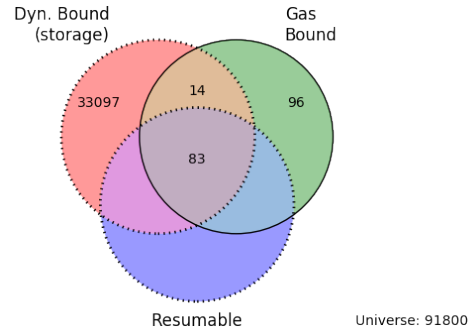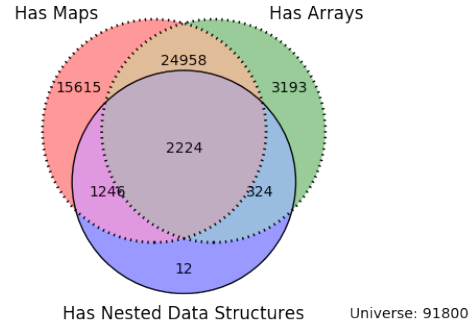
(b) Prevalence of loops bound by gas, resumable loops and loops dynamically bound by storage state.

Fig. 11. Classification of loops.

(a) Prevalence of loops as a way to access arrays. Contracts that have arrays on storage tend to iterate on them, usually in explicit loops.

(b) Prevalence of maps, arrays and nested data structures.

Fig. 12. Classification of data structure use.

Many contracts have nested data structures. Nested data structures are quite popular in Ethereum smart contracts and Figure 12b illustrates the importance of reverse-engineering these data structures from bytecode as part of the analysis.

6.4 Threats to validity and clarifications

There are a number of threats to validity and clarifications of our experimental evaluation which are worth mentioning:

Difficulty in establishing the base truth. Unlike other popular "static analysis for security" topics, such as Android taint analysis, smart contracts are a relatively new technology. As such, no labeled contracts are available for benchmarking, and the vulnerabilities we are demonstrating in this work have not been published. Manually inspecting contracts is hard and error-prone, taking at least 1 to 2 hours per contract to establish whether a flagged vulnerability is indeed admissible. In addition, many contracts on the blockchain do not have source code. We have partially addressed

these threats by manually inspecting contracts with sources to reduce manual errors. We think that our work can be part of the solution to this problem by releasing a small, labeled benchmark suite based on the manually inspected contracts.

Bias. We may be introducing an element of bias by only inspecting contracts that have source code available online. This sample set may either be more sophisticated and tested than the average contract on the blockchain, or, conversely, toys, used for learning and exposition. The contracts we chose to manually inspect hold very little Ether. However, this is a natural consequence of the Pareto distribution of Ether between different contracts. Only 1195 contract programs (out of the 90k+) hold more than 1 Ether, and one (1) contract program deployed over 43 instances holds 28% of all the Ether held by all the contracts.

Consequences of vulnerabilities. Although we claim that the contracts that are flagged as vulnerable hold more than the equivalent of $5B, this does not mean that: (1) exploiting these vulnerabilities, e.g., to block a contract, is easy or cheap, or (2) the vulnerability blocks all Ether in a contract. For instance, it takes resources to execute the unbounded mass operation vulnerability, as it takes an order of magnitude as much gas to add an element to an array than to iterate over one more array element.

7 RELATED WORK

Analysis and verification for smart contracts have received substantial attention recently due to the security issues inherent in the high-risk paradigm of smart contract development. A precursor of this work is Michael Kong's thesis [Kong 2017] that has been based on Vandal [Various 2018]. The thesis introduces various types of gas vulnerabilities and expoits and provides also algorithms to compute loop bounds which expose gas vulnerabilities. The Vandal framework is an open-source framework for decompiling EVM bytecode and analysing the bytecode via logic specification expressed in Soufflé [Jordan et al. 2016].

Overall, our approach is distinguished by being fully static (i.e., more exhaustive, yet possibly with false positives), applicable to the EVM bytecode level, and focused on modeling data structures and high-level gas vulnerabilities.

Previous works can be classified according to their underlying techniques, including symbolic execution, formal verification, and abstract interpretation. Systems including Oyente [Luu et al. 2016a], MAIAN [Nikolic et al. 2018], GASPER [Chen et al. 2017] and recent work [Grossman et al. 2017] use a symbolic execution/trace semantics approach that is fundamentally unsound, since only some program paths of smart contracts can be explored.

Semi-automated *formal verification* approaches have also been proposed [Amani et al. 2018; Bhargavan et al. 2016; Grishchenko et al. 2018; Hildenbrandt et al. 2017; Hirai 2017; Why3 2018] for performing complete analyses of smart contracts using interactive theorem provers such as Isabelle/HOL [Isabelle 2018], F* [FStarLang 2018], Why3 [Why3 2018], and $\mathbb{K}$ [K Framework 2018]. These approaches have a common theme: a formal model of a smart contract is constructed and mathematical properties are shown via the use of a semi-automated theorem prover. Recently a complete small-step semantics of EVM bytecode has been formalized for the F* proof assistant [Grishchenko et al. 2018]. Other systems such as KEVM [Hildenbrandt et al. 2017] use the $\mathbb{K}$ framework based on reachability logic. Due to their reliance on semi-automated theorem provers which require substantial manual intervention for proof construction, these formal verification approaches do not scale for analyzing the millions of smart contracts currently deployed on the blockchain.

In contrast to formal verification work for smart contracts, abstract interpretation approaches [Kalra et al. 2018; Mavridou and Laszka 2018] do not require human intervention; however, they introduce false-positives. The Zeus framework [Kalra et al. 2018] translates Solidity source

code to LLVM [LLVM 2018] before performing the actual analysis in the SeaHorn verification framework [SeaHorn 2018]. An alternative approach is that of [Mavridou and Laszka 2018], in which Solidity code is abstracted to finite-state automata. Our approach is also partly that of abstract interpretation. However, in contrast to existing abstract interpretation frameworks, our analysis is performed directly on EVM bytecode. For this purpose, we built a decompiler that translates EVM bytecode to an analyzable form.

Various exploits have been broadly identified in the literature [Atzei et al. 2016; Bartoletti et al. 2017; Delmolino et al. 2015; Sergey and Hobor 2017]: exploits related to Solidity, the EVM and the blockchain itself. These exploits have been highlighted by the community outside of publications. For instance, the company Consensys [Consensys 2018a] maintains a website [Consensys 2018b] outlining the exploits mentioned in the literature, as well as additional exploits such as data overflows and underflows, and suggestions to write better smart contracts (such as isolating external calls into their own transactions, instead of executing them in a single transaction to minimize the risk of failures and side-effects).

Oyente [Luu et al. 2016b] identifies four exploits: Transaction-Ordering Dependence, Timestamp Dependence, exceeding the call stack limit of 1024 (Callstack attack) and reentrancy. Transaction-ordering dependence refers to issues of concurrency where multiple transactions are required to be executed in a particular order, but a miner can manipulate the transaction order. The formal verification tool by [Bhargavan et al. 2016] detects three classes of vulnerabilities, two of which were covered by Oyente. These include checking the return value of external address calls, and reentrancy. However, an upper bound analysis on gas required for a given transaction was created. These patterns were verified in F^* by translating the contracts into F^* code, from which patterns were applied to detect vulnerabilities. Similarly, the FSolidM framework [Mavridou and Laszka 2018] checks for reentrancy and transaction ordering vulnerabilities. It can also detect coding patterns such as time constraint and authorization issues as outlined in [Bartoletti et al. 2017]. The MAIAN framework [Nikolic et al. 2018] focuses on finding vulnerabilities in smart contracts such as locking of funds indefinitely, leaking funds to arbitrary users, and smart contracts that can be killed by anyone. The GASPER [Chen et al. 2017] identifies GAS-costly programming patterns. The ZEUS system [Kalra et al. 2018] conducts policy checking for a set of policies including reentrancy, unchecked send, failed send, integer overflow, transaction state dependence/order and block state dependence. Grossman et al. [2017] focus exclusively on detecting non-callback-free contracts.

8 CONCLUSIONS

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 6.6 million 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.

As part of future work we will be investigating how further improvements can be made to the decompilation infrastructure to further increase precision and scalability, as well as how different semantic aspects of smart contracts can be modeled to identify more and unexpected vulnerabilities.

ACKNOWLEDGMENTS

REFERENCES

Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. 2018. Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2018)*. ACM, New York, NY, USA, 66–77. https://doi.org/10.1145/3167084

Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2016. *A survey of attacks on Ethereum smart contracts*. Technical Report. Cryptology ePrint Archive: Report 2016/1007, https://eprint.iacr.org/2016/1007.

Massimo Bartoletti, Salvatore Carta, Tiziana Cimoli, and Roberto Saia. 2017. Dissecting Ponzi schemes on Ethereum: identification, analysis, and impact. (2017).

Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. 2016. Formal Verification of Smart Contracts: Short Paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security (PLAS '16)*. ACM, New York, NY, USA, 91–96. https://doi.org/10.1145/2993600.2993611

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proc. of the 24th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA '09)*. ACM, New York, NY, USA.

Vitalik Buterin. 2013. A Next-Generation Smart Contract and Decentralized Application Platform. https://github.com/ethereum/wiki/wiki/White-Paper. (2013).

T. Chen, X. Li, X. Luo, and X. Zhang. 2017. Under-optimized smart contracts devour your money. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 442–446. https://doi.org/10.1109/SANER.2017.7884650

Consensys. 2018a. Consensys logo. (2018). https://new.consensys.net/ Accessed: 2018-04-17.

Consensys. 2018b. Ethereum Smart Contract Best Practices. (2018). https://consensys.github.io/smart-contract-best-practices/ Accessed: 2018-04-17.

Kevin Delmolino, Mitchell Arnett, Ahmed E. Kosba, Andrew Miller, and Elaine Shi. 2015. Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab. *IACR Cryptology ePrint Archive* 2015 (2015), 460.

etherscan.io. [n. d.]. Ethereum Contracts with Verified Source Codes. ([n. d.]). https://etherscan.io/contractsVerified Accessed: 2018-03-15.

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended static checking for Java. In *Proc. of the ACM SIGPLAN 2002 Conf. on Programming Language Design and Implementation*. ACM Press, 234–245.

FStarLang. 2018. F*: A Higher-Order Effectful Language Designed for Program Verification. (2018). https://www.fstar-lang.org/ Accessed: 2018-04-17.

Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum smart contracts. *CoRR* abs/1802.08660 (2018). arXiv:1802.08660 http://arxiv.org/abs/1802.08660

Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2017. Online Detection of Effectively Callback Free Objects with Applications to Smart Contracts. *Proc. ACM Program. Lang.* 2, POPL, Article 48 (Dec. 2017), 28 pages. https://doi.org/10.1145/3158136

Everett Hildenbrandt, Xiaoran Zhu, and Nishant Rodrigues. 2017. KEVM: A Complete Semantics of the Ethereum Virtual Machine.

Yoichi Hirai. 2017. Defining the Ethereum Virtual Machine for Interactive Theorem Provers. In *Financial Cryptography and Data Security*, Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y.A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson (Eds.). Springer International Publishing, Cham, 520–535.

Marco Iansiti and Karim R. Lakhani. 2017. The Truth about Blockchain. *Harvard Business Review* 95 (Jan. 2017), 118–127. Issue 1.

Neil Immerman. 1999. *Descriptive Complexity*. Springer. https://doi.org/10.1007/978-1-4612-0539-5

Isabelle. 2018. Isabelle. (2018). https://isabelle.in.tum.de/ Accessed: 2018-04-17.

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.

K Framework. 2018. K Framework. (2018). http://www.kframework.org/index.php/Main_Page Accessed: 2018-04-17.

Sukrit Kalra, Seep Goel, Seep Goel, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. (2018).

Michael Kong. 2017. A Scalable Method to Analyze Gas Costs, Loops and Related Security Vulnerabilities on the Ethereum Virtual Machine. https://github.com/usyd-blockchain/vandal/wiki/pubs/MKong17.pdf. (11 2017).

Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Sam Guyer, Uday Khedker, Anders Møller, and Dimitrios Vardoulakis. 2014. In Defense of Soundiness: A Manifesto. *Commun. ACM* (2014), to appear.

LLVM. 2018. The LLVM Compiler Infrastructure Project. (2018). https://llvm.org/ Accessed: 2018-04-17.

Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016a. 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

Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016b. 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. (2018). http://aronlaszka.com/papers/mavridou2018designing.pdf

Mayur Naik. 2011. Chord: A Versatile Platform for Program Analysis. In *2011 ACM SIGPLAN Conf. on Programming Language Design and Implementation*. Tutorial.

Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. 2009. Effective static deadlock detection. In *Proc. of the 31st International Conf. on Software Engineering (ICSE '09)*. ACM, New York, NY, USA, 386–396. https://doi.org/10.1109/ICSE.2009.5070538

Satoshi Nakamoto. 2009. Bitcoin: A Peer-to-Peer Electronic Cash System. https://www.bitcoin.org/bitcoin.pdf. (2009).

Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. *CoRR* abs/1802.06038 (2018). arXiv:1802.06038 http://arxiv.org/abs/1802.06038

SeaHorn. 2018. SeaHorn | A Verification Framework. (2018). http://seahorn.github.io/ Accessed: 2018-04-17.

Ilya Sergey and Aquinas Hobor. 2017. A Concurrent Perspective on Smart Contracts. *CoRR* abs/1702.05511 (2017). arXiv:1702.05511 http://arxiv.org/abs/1702.05511

Olin Shivers. 1991. *Control-Flow Analysis of Higher-Order Languages*. Ph.D. Dissertation. Carnegie Mellon University.

Olin Shivers. 2004. Higher-order control-flow analysis in retrospect: lessons learned, lessons abandoned. In *Best of PLDI 1988*, Kathryn S. McKinley (Ed.), Vol. 39. 257–269.

H. Thielecke. 1999. Continuations, functions and jumps. *ACM SIGACT News* 30 (Jan. 1999), 33–42. Issue 2.

Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In *Proc. of the 1999 Conf. of the Centre for Advanced Studies on Collaborative research (CASCON '99)*. IBM Press, 125–135. http://dl.acm.org/citation.cfm?id=781995.782008

Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. 2000. Optimizing Java Bytecode Using the Soot Framework: Is It Feasible?. In *Proc. of the 9th International Conf. on Compiler Construction (CC '00)*. Springer, 18–34.

Various. [n. d.]a. GovernMental page. ([n. d.]). http://governmental.github.io/GovernMental/ Accessed: 2018-04-14.

Various. [n. d.]b. Safety - Ethereum Wiki. https://github.com/ethereum/wiki/wiki/Safety. ([n. d.]). Accessed: 2018-04-15.

Various. 2018a. Documentation for the LLL compiler – LLL Compiler Documentation 0.1 documentation. (2018). http://lll-docs.readthedocs.io/en/latest/index.html Accessed: 2018-04-17.

Various. 2018b. GitHub - ethereum/serpent. (2018). https://github.com/ethereum/serpent Accessed: 2018-04-17.

Various. 2018c. GitHub - ethereum/solidity: The Solidity Contract-Oriented Programming Language. (2018). https://github.com/ethereum/solidity Accessed: 2018-04-17.

Various. 2018d. GitHub - ethereum/vyper: New experimental programming language. (2018). https://github.com/ethereum/vyper Accessed: 2018-04-17.

Various. 2018. Vandal – A Static Analysis Framework for Ethereum Bytecode. (2018). https://github.com/usyd-blockchain/vandal/ Accessed: 2018-07-30.

Peter Vessenes. 2016. Ethereum Griefing Wallets: Send w/Throw Is Dangerous. (2016). http://vessenes.com/ethereum-griefing-wallets-send-w-throw-considered-harmful

Why3. 2018. Why3. (2018). http://why3.lri.fr/ Accessed: 2018-04-17.

Gavin Wood. 2014. Ethereum: A Secure Decentralised Generalised Transaction Ledger. http://gavwood.com/Paper.pdf. (2014).