Smart Contract Proxy Analysis

Review Report

Tuesday 31st May, 2022

The evolution of Smart Contract protocols both in respect to size and complexity has led to the creation of new design patterns, centered around modularity, maintainability and upgradeability. Many of these design patterns such as the **Proxy Upgrade** pattern, or the **Diamond** pattern, leverage `delegatecall` to implement their core functionality.

Current state-of-the art static analysis tools do not take into account the unique intricacies of having shared mutable state across multiple smart contracts, provided by the `delegatecall` opcode.

This study introduces a general technique for multi-contract analysis under `delegatecall` , through the modularisation of the Gigahorse analysis framework and the propagation of storage facts between smart contracts during analysis execution. Following this we present a new tool called **SOuL-Splitter**, which generates multi-contract evaluation test sets through automated decomposition of existing smart contracts.

Overall we find that our analysis technique is highly effective, with some vulnerabilities exhibiting over a 70 point improvement in recall as compared with their single contract counterparts.

1 Introduction

Ethereum [1] is a blockchain platform designed to support the execution of Turing complete programs known as Smart Contracts, through the use of a "decentralised" virtual machine called the Ethereum Virtual Machine (EVM).

Emerging design patterns in the Ethereum smart contract space, such as the **Proxy Upgrade** or **Diamond**, are leveraging the `delegatecall` opcode to solve the scaling concerns of modularity, maintainability and upgradeability.

Current analysis tooling has not yet caught up with developers, many of the state-of-the-art tools (Securify [2], Oyente [3], Slither [4]) do not properly model the security implications of `delegatecall`. This leaves developers leveraging `delegatecall` to have an incomplete view of the possible vulnerabilities. This is especially important for the complex call topologies used in the Diamond pattern, or in incompatibilities between implementation versions for standard proxies.

Our work solves this gap in the static analysis space by modifying the Gigahorse [5] analysis framework, and introducing a general technique for multi-contract analysis under `delegatecall` which leverages analysis componentisation. The original contributions to the field can be summarised as follows;

- Generalised solution for multi-contract analysis implementation and adaptation, including:
  - Adaptation of the Ethainter [6] Taint Analysis
  - Adaptation of the MadMax [7] Denial of Service Analysis
  - Adaptation of Symvalic Analysis [8]
- Automated test set generation for multi-contract analysis evaluation

2 Background & Literature Overview

2.1 Ethereum, the EVM and Smart Contracts

The Ethereum Virtual Machine (EVM) is a stack based virtual machine with a 256-bit word size, a stack size of 1024 elements, cheap non-persistent memory, and expensive persistent storage [9]. The EVM provides standard opcodes for stack based virtual machines, as well as blockchain specific opcodes such as `sha3` or `blockhash`.

Smart contracts communicate internally through the use of the `CALL` family of opcodes, which create a new execution context (stack and memory) for the target smart contract and allow it to execute before returning control to the caller. During execution smart contracts read input from a special region known as the `calldata` buffer and can return data from their own memory to the `returndata` buffer accessible by the caller contract.

The primary opcode of interest for this study is the `delegatecall` [10] opcode, which issues a transaction in which the callee contract executes with the state of the caller contract, and all state modifications made are reflected back into the state of caller.

2.2 Gigahorse

The Gigahorse [5] analysis framework used for this study is an EVM bytecode decompiler and binary lifter written in Souffle Datalog, designed to produce a high level Three Address Code (TAC) and Datalog fact sets to support the implementation of static analyses.

The binary lifting performed by Gigahorse revolves around the reversal of code size optimisations performed by the Solidity compiler, as these optimisations cause very complex control flow graphs which are not desirable for static analysis. The primary mechanism used by the Solidity compiler for reducing code size is the reuse of code chunks, this is achieved by creating jump points to common sequences of instructions. One can draw an analogy between this and the LZ family of algorithms for text compression [11].

2.2.1 Ethainter

Ethainter [6] is a taint analysis built on top of the Gigahorse framework, with the specific goal of detecting privilege escalations or critical section manipulation.

For EVM bytecode the **sources** of external data are attributed to either a `calldataload` or a `calldatacopy`. Critical code sections, or **sinks**, are defined as operations which can transfer value, variables used in guards or the target of a `delegatecall` (which allows for "code injection").

Ethainter analyses the propagation of data from **sources** to **sinks** through unguarded control flow paths. During analysis runtime the list of unguarded control flow paths is not constant, as Ethainter can detect tainted guard variables and open new control flow paths. The analysis defines a total of 11 rules of inference which are used to determine tainted guards, storage reads/writes, and general tainted sinks. A further 6 rules of inference are defined to detect data structures.

2.2.2 MadMax

MadMax [7] was the first analysis written for the Gigahorse framework, with the aim of detecting **out-of-gas** exceptions in Smart Contracts. **Out-of-gas** exceptions are triggered by the EVM whenever a transaction sender sends insufficient funds to pay for the gas used during the execution of the transaction.

The primary vulnerability of focus for this study is **UnboundedMassOp**. **UnboundedMassOp** detects the pattern of public methods which can monotonically increase the number of items in an array, and another public method which iterates over the same array. This pattern allows adversaries to launch a Denial of Service (DOS) attack on the second function by writing arbitrary data into the array.

2.2.3 Symvalic Analysis

Symvalic Analysis [8] is currently the most advanced analysis written on top of the Gigahorse framework. It was written with the goal of bridging the completeness of Static analysis with the precision of formal verification methods. This is achieved through the utilisation of symbolic evaluation in combination with concrete values, as well as the use of record tables to implement loose coupling between values.

Symvalic Analysis is not an analysis onto itself, rather it provides a high level framework to implement analysis clients on top of. For the focus of this study the analyses used will be limited to the Symvalic equivalents of Ethainter and MadMax.

3 Core & Implementation

3.1 Multi Contract Analysis

The final implementation for multi-contract analysis leverages Souffle components to encapsulate analysis, and implements cross contract functionality through careful propagation of storage related facts. This design allows for a well curated shared state which increases the precision of the analysis, and supports the re-use of existing analysis. Another benefit of this approach is the ability to tailor decompilation for each contract in the analysis independently, with smaller contracts being more aggressively in-lined and decompiled, and larger contracts which present scaling issues being less aggressively decomplied.

3.1.1 Componentisation

The componentisation approach is best illustrated in listing 1, which shows Ethainter being applied to two contracts. An assumption for this approach is that all the public functions in each contract are reachable, if this is not the case then extra reachability constraints can be added to any analysis.

3.1.2 Fact Propagation

Fact propagation is unique to every analysis, and requires the propagated facts to be rooted in storage changes. This may require refactoring certain analysis to encapsulate storage changes better, however this has not been required in practice.

Ethainter can be converted in to a multi-contract analysis by propagation a single relation, as shown in listing 1. The relation `GlobalVariableModifiableByAttacker` identifies storage locations which can be tainted by external sources. Sharing this information is essentially introducing new taint sources to the other contracts, which can lead to the discovery of more tainted storage locations.

```
#define MULTI_CONTRACT
#include "gigahorse-toolchain/souffle-addon/functor_includes.dl"

.comp Contract{
      #include "ethainter.dl"
}

.init contract1 = Contract
.init contract2 = Contract

contract1.GlobalVariableModifyableByAttacker(as(stmt,
 ↪  contract1.Statement), as(globalVal, contract1.Value)):-
          contract2.GlobalVariableModifyableByAttacker(stmt,
           ↪  globalVal).

contract2.GlobalVariableModifyableByAttacker(as(stmt,
 ↪  contract2.Statement), as(globalVal, contract2.Value)):-
          contract1.GlobalVariableModifyableByAttacker(stmt,
           ↪  globalVal).
```

Listing 1: Two Contract Ethainter Analysis

3.1.3 Automated Test Set Generation

In order to quantify the validity of the multi-contract analysis approach we needed a test set of vulnerable multi-contract protocols. Unfortunately relying on real world multi-contract exploits would not have produced a large enough evaluation set, as such test set generation was required.

The approach taken for the evaluation metrics was to compare the multi-contract analysis against its single contract counterpart. This was achieved by creating a new tool called **SOuL-Splitter**, which takes smart contracts and decomposes their functionality across multiple implementation smart contracts.

SOuL-Splitter works by generating an Abstract Syntax Tree (AST) for a solidity source file and traversing it for `public` functions. Changing the visibility of all `public` functions to `internal` essentially removes them in the bytecode as the Solidity compiler will perform dead code elimination. This "blank" slate is then used to generate the new *sub-contracts* through selectively re-enabling functions, by setting their visibility level back to `public`. This process is currently performed for up to 15 public functions after which functions are grouped together. A benefit to this approach is the handling of mutually dependent `public` functions, as it allows for the generation of *sub-contracts* where only one `public` function is visible at a time whilst still preserving full functionality.

4 Evaluation

The evaluation of the multi-contract analysis was performed using the Smartbugs-Wild contract dataset. The dataset contains over 47,398 real world contracts extracted from the Ethereum mainnet, which were then converted into multi-contracts using SOuL-Splitter introduced in section 3.1.3. Using this dataset as the ground truth, the single and split versions of each analysis were run in order to generate the recall scores. This method produced a dataset of 1026 contracts for MadMax, 1248 contracts for Ethainter, and 1520 for Symvalic.

4.1 Results

Vulnerability	**Single**	**Multi**	**Missed**	**Recall**
OverflowLoopIterator	468	468	0	100%
UnboundedMassOp	305	596	23	96.28%
WalletGriefing	98	98	0	100%

Table 1: Multi-Madmax Evaluation

Vulnerability	**Single**	**Multi**	**Missed**	**Recall**
AccessibleSelfdestruct	94	213	3	98.61%
TaintedDelegatecall	13	13	0	100%
TaintedERC20Transfer	2	2	0	100%
TaintedOwnerVariable	178	586	44	93.02%
TaintedSelfdestruct	58	144	1	99.31%
TaintedValueSend	174	453	100	81.92%

Table 2: Multi-Ethainter Evaluation

Vulnerability	**Single**	**Multi**	**Missed**	**Recall**
AccessibleSelfDestruct	156	156	0	100%
CallToTaintedFunction	9	45	1	97.83%
Griefing	67	67	0	100%
Reentrancy	27	112	3	97.39%
TaintedDelegateCall	5	5	0	100%
TaintedOwnershipGuard	17	17	0	100%
TaintedSelfDestruct	29	29	0	100%
UnboundedIteration	403	1182	20	98.34%

Table 3: Multi-Symvalic Evaluation

4.2 Discussion

The results for Madmax at first appear somewhat underwhelming, however it should be noted that the **UnboundedMassOp** is the only vulnerability that is possibly dependent on multiple public functions. Taking **UnboundedMassOp** on its own shows a roughly 63 point improvement for multi-contract analysis over split-single contract analysis.

Most the results of multi-contract Ethainter are extremely promising, with four of the six analyses improving at least 60 points. The result for **TaintedOwnerVariable** is the most promising, with an over 70 point improvement in recall. **TaintedValueSend** shows a significant decrease in recall, however this is can be attributed to the increased precision rather than a limitation of the multi contract approach. Examining 10 of the 44 missed **TaintedOwnerVariable** reveals that all the contracts were false positives, as they leverage an extra function call in their modifier guards which is known to cause problems with the analysis. The simplified Control Flow Graphs (CFGs) in the split contracts allowed for better inlining which removed the function call, and allowed the analysis to detect the guard.

The results for Symvalic are extremely promising, with its equivalent of **UnboundedMassOp** exhibiting a 72 point improvement, almost 10 points more than its non symbolic counterpart. Symbolic re-entrancy is another very promising result, especially within the context of the other multi-contract analysis Clairvoyance [12] being solely limited to this. The symbolic equivalents to Ethainter are less promising in this result set, however it should be noted that this test set was limited to a 6 contract split rather than the 15 contract split used in the other evaluations. The change in splitting came down to a scalability problem with the compiler, as Symvalic analysis is significantly more complex than previous analysis. The compiled binary for 6 contract Symvalic analysis is roughly on par with 15 contract Ethainter, with respect to size and compilation time.

5 Conclusion

We presented a generalised approach towards multi-contract analysis under `delegatecall`, within the context of the Gigahorse analysis framework and the Souffle language.

The main criticism of this body of work is its limitation to an artificial dataset, and re-evaluation of this approach with real world contracts should better highlight its strengths and weaknesses.

The next steps for this analysis technique are in its application to real world contracts, as well as testing uni-directional fact propagation for standard proxy upgrades. Following this another key research area is determining how to extend this technique to standard calls, which would enable enhanced precision for analysis through the propagation of `calldata`.

Overall the componentisation technique proves to be a simple, yet impressively effective, method of converting existing single contract analysis to multi-contract analysis.

References

[1] V. Buterin, "A next-generation smart contract and decentralized application platform," *Ethereum White Paper*, 2013. [Online]. Available: https://ethereum.org/en/whitepaper/

[2] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 67–82. [Online]. Available: https://doi.org/10.1145/3243734.3243780

[3] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Oyente: Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 254–269. [Online]. Available: https://doi.org/10.1145/2976749.2978309

[4] J. Feist, G. Greico, and A. Groce, "Slither: A static analysis framework for smart contracts," in *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*, ser. WETSEB '19. IEEE Press, 2019, p. 8–15. [Online]. Available: https://doi.org/10.1109/WETSEB.2019.00008

[5] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis, "Gigahorse: Thorough, declarative decompilation of smart contracts," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 1176–1186. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00120

[6] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis, "Ethainter: A smart contract security analyzer for composite vulnerabilities," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 454–469. [Online]. Available: https://doi.org/10.1145/3385412.3385990

[7] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Analyzing the out-of-gas world of smart contracts," *Commun. ACM*, vol. 63, no. 10, p. 87–95, sep 2020. [Online]. Available: https://doi.org/10.1145/3416262

[8] Y. Smaragdakis, N. Grech, S. Lagouvardos, K. Triantafyllou, and I. Tsatiris, "Symbolic value-flow static analysis: Deep, precise, complete modeling of ethereum smart contracts," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: https://doi.org/10.1145/3485540

[9] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum yellow paper*, 2021. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

[10] V. Buterin, "Eip-7: Delegatecall," 2015. [Online]. Available: https://eips.ethereum.org/EIPS/eip-7

[11] Welch, "A technique for high-performance data compression," *Computer*, vol. 17, no. 6, pp. 8–19, 1984.

[12] Y. Xue, M. Ma, Y. Lin, Y. Sui, J. Ye, and T. Peng, "Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1029–1040. [Online]. Available: https://doi.org/10.1145/3324884.3416553