# Using Automated Reasoning Techniques for Enhancing the Efficiency and Security of (Ethereum) Smart Contracts

Elvira Albert[1,2(✉)], Pablo Gordillo[1], Alejandro Hernández-Cerezo[1], Clara Rodríguez-Núñez[1], and Albert Rubio[1,2]

[1] Complutense University of Madrid, Madrid, Spain
[2] Instituto de Tecnología del Conocimiento, Madrid, Spain
elvira@fdi.ucm.es

The use of the Ethereum blockchain platform [17] has experienced an enormous growth since its very first transaction back in 2015 and, along with it, the verification and optimization of the programs executed in the blockchain (known as Ethereum *smart contracts*) have raised considerable interest within the research community. As for any other kind of programs, the main properties of smart contracts are their *efficiency* and *security*. However, in the context of the blockchain, these properties acquire even more relevance. As regards efficiency, due to the huge volume of transactions, the cost and response time of the Ethereum blockchain platform have increased notably: the processing capacity of the transactions is limited and it is providing low transaction ratios per minute together with increased costs per transaction. Ethereum is aware of such limitations and it is currently working on solutions to improve scalability with the goal of increasing its capacity. As regards security, due to the public nature and immutability of smart contracts and the fact that their public functions can be executed by any user at any time, programming errors can be exploited by attackers and have a high economic impact [7,13]. Verification is key to ensure the security of smart contract's execution and provide safety guarantees. This talk will present our work on the use of automated reasoning techniques and tools to enhance the security and efficiency [2–4,6] of Ethereum smart contracts along the two directions described below.

*Security.* Our main focus on security will be to detect and avoid potential *reentrancy* attacks, one of the best known and exploited vulnerabilities that have caused infamous attacks in the Ethereum ecosystem due to they economic impact [9,11,15]. Reentrancy attacks might occur on programs with callbacks, a mechanism that allows making calls among contracts. Callbacks occur when a method of a contract invokes a method of another contract and the latter, either directly or indirectly, invokes one or more methods of the former before the original method invocation returns. While this mechanism is useful and powerful

This work was funded partially by the Ethereum Foundation (Grant FY21-0372), the Spanish MCIU, AEI and FEDER (EU) project RTI2018-094403-B-C31 and by the CM project S2018/TCS-4314 co-funded by EIE Funds of the European Union.

© The Author(s) 2022
J. Blanchette et al. (Eds.): IJCAR 2022, LNAI 13385, pp. 3–7, 2022.
https://doi.org/10.1007/978-3-031-10769-6_1

in event-driven programming, it has been used to exploit vulnerabilities. Our approach to detect potential reentrancy problems is to ensure that the program meets the Effectively Callback Freeness (ECF) property [10]. ECF guarantees the modularity of a contract in the sense that executions with callbacks cannot result in new states that are not reachable by callback free executions. This implies that the use of callbacks will not lead to unpredicted, potentially dangerous, states. In order to ensure the ECF property, we use commutation and projection of fragments of code [6]. Intuitively, given a function fragment $A$ followed by $B$ (denoted $A.B$), in case we can receive a callback to some function $f$ between these fragments (that is, $A.f.B$), we ensure safety by proving that this execution that contains callbacks is equivalent to a callback free execution: either to $A.B$ (projection), $f.A.B$ (left-commutation) or $A.B.f$ (right-commutation). The use of automated reasoning techniques enables proving this kind of properties. Inspired by the use of SMT solvers to prove redundancy of concurrent executions [1,8,16], we have implemented such checks using state-of-the-art SMT solvers.

The ECF property can be generalized to allow callbacks to introduce new behaviors as long as they are benign, as [5] does by defining the notion of R-ECF. The main difference between ECF and R-ECF is that while ECF checks that the states reached by executions with callbacks are exactly the same as the ones reached by executions that do not contain callbacks, R-ECF checks that they satisfy a relation with respect to the states reached without callbacks. This way, R-ECF is able to recognize and distinguish the benign behaviors introduced by callbacks from the ones that are potentially dangerous, while ECF cannot. The main application of R-ECF is that, from a particular invariant of the program, it allows reducing the problem of verifying the invariant in the presence of callbacks, to the callback-free setting. For example, if we consider the invariant $\texttt{balance} \geq 0$ and prove that the contract is R-ECF with respect to the relation $\texttt{balance}_{cb} \geq \texttt{balance}_{cbfree}$ (i.e., the balance reached by executions with callbacks is greater than the one reached without callbacks), then we only need to consider callback free executions in order to prove the preservation of the invariant.

We considered as benchmarks the top-150 contracts based on volume of usage, and studied the modularity of their functions in terms of ECF and R-ECF. A total of 386 of their functions were susceptible to have callbacks, from which 62.7% were verified to be ECF. The R-ECF approach was able to increase the accuracy of the analysis, being able to prove the correctness of an extra 2% of functions [5,6].

*Efficiency.* The main focus on efficiency will be on optimizing the resource consumption of smart contract executions. On the Ethereum blockchain, the resource consumption is measured in terms of *gas*, a unit introduced in the system to quantify the computational effort and charge a fee accordingly in order to have a transaction executed. To understand how we can optimize gas, we need to discuss it (and do it) at the level of the Ethereum bytecode. Smart contracts in Ethereum are executed using the Ethereum Virtual Machine (EVM). The EVM is a simple stack-based architecture which uses 256-bit words and has its own repertory of instructions (EVM opcodes). In the EVM, the mem-

ory model is split into two different structures: the *storage*, which is persistent between transactions and expensive to use; and the *memory*, which does not persist between transactions and is cheaper. Each opcode has a gas cost associated to its execution. Besides, an additional fee must be paid for each byte when the smart contract is deployed. Thus, the resource to be optimized can be either the total amount of gas in a program or its size. Even though both criteria are usually related, there are some situations in which they do not correlate. For instance, pushing a big number in the stack consumes a small amount of gas and increases significantly the bytecode size, whereas obtaining the same value using arithmetic operations is more expensive but involves fewer bytes.

Among all possible techniques to optimize code, we have used the technique known as superoptimization [12]. The main idea of superoptimization is automatically finding an equivalent optimal sequence of instructions to another given loop-free sequence. In order to achieve this goal, we enumerate all possible candidates and determine the best option among them *wrt.*the optimization criteria. In the context of EVM, there exists several superoptimizers: EBSO [14], SYRUP [3,4] and GASOL [2]. The techniques presented in this work correspond to the ones implemented in GASOL, which are an improvement and extension of the ones in SYRUP. We apply two kinds of automated reasoning techniques to superoptimize Ethereum smart contracts, symbolic execution and Max-SMT as described next.

- Symbolic execution is used to obtain a a representation on how the stack and memory evolves *wrt.* to an initial stack. We determine the lowest size of the stack needed to perform all the operations in a block and apply symbolic execution to an initial stack containing that number of unknown stack variables. Opcodes representing operations that don't manage the stack are left as uninterpreted functions. Then, we apply as many simplification rules as possible from a fixed set of rules. Depending on the chosen criteria, some rules are disabled if they lead to worse candidates. Moreover, we apply static analysis regarding memory opcodes to determine whether there are some redundant store or load operations inside a block that can be safely removed or replaced. This leads to a simplified specification of the optimal block.
- The second technique involves synthesizing the optimal block from a given symbolic representation using a Max-SMT solver. The synthesis problem is expressed as a first-order formula in which every model corresponds to a valid equivalent block. Our encoding is expressed in the simple logic *QF_IDL*, so that the Max-SMT solver can reason effectively on EVM blocks. In this encoding, the length of the sequence of instructions is fixed by an upper bound so that quantifiers are avoided. NOP operations are considered in the encoding to allow shorter sequences. The state of the stack is represented explicitly for each position in the sequence. Every instruction in the block and every basic stack operation have a constraint that reflects the impact they have on the stack for each possible position. Memory accesses are encoded as a partial order relation that synthesizes the dependencies among them. Regarding the optimization process, we express the cost (gas or bytes-size) of

each instruction using soft constraints. For both criteria, the corresponding set of soft constraints satisfies that an optimal model returned by the solver corresponds to an optimal block for that criteria.

Combining both approaches, we obtain significant savings for both criteria. For a subset of 30 smart contracts, selected among the latest published in Etherscan as of June 21, 2021 and optimized using the compiler solc v0.8.9, GASOL still manages to reduce 0.72% the amount of gas with the gas criteria enabled, and decreases the overall size by 3.28% with the size criteria enabled.

*Future work.* The current directions for future work include enhancing the performance of the smart contract optimizer in both accuracy and scalability of the process while keeping the efficiency. For the accuracy we are currently working on adding further reasoning on non-stack operations while staying in a quite simple logic. This will allow us to consider a wider set of equivalent blocks and hence increase the savings. Scalability can be threatened when we consider blocks of code of large size. We are investigating different approaches to scale better, including heuristics to partition the blocks in smaller sub-blocks, more efficient SMT encodings, among others. Finally, another direction for future work is to formally prove the correctness of the optimizer, *i.e.*developing a checker that can formally prove the equivalence of the optimized and the original (Ethereum) bytecode. For this, we are planning to use the Coq proof assistant in which we will develop a checker that, given an original bytecode –that corresponds a block of the control flow graph– and its optimization, it can formally prove their equivalence for any possible execution, and optionally it can generate a soundness proof that can be used as certificate.

## References

1. Albert, E., Gómez-Zamalloa, M., Isabel, M., Rubio, A.: Constrained dynamic partial order reduction. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 392–410. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_24
2. Albert, E., Gordillo, P., Hernández-Cerezo, A., Rubio, A.: A Max-SMT superoptimizer for EVM handling memory and storage. In: Fisman, D., Rosu, G. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2022. LNCS, vol. 13243. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_11
3. Albert, E., Gordillo, P., Hernández-Cerezo, A., Rubio, A., Schett, M.A.: Superoptimization of smart contracts. ACM Trans. Softw. Eng. Methodol. (2022)
4. Albert, E., Gordillo, P., Rubio, A., Schett, M.A.: Synthesis of super-optimized smart contracts using Max-SMT. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12224, pp. 177–200. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53288-8_10
5. Albert, E., Grossman, S., Rinetzky, N., Nunez, C.R., Rubio, A., Sagiv, M.: Relaxed effective callback freedom: a parametric correctness condition for sequential modules with callbacks. IEEE Trans. Dependable Secure Comput. (2022)

6. Albert, E., Grossman, S., Rinetzky, N., Rodríguez-Núñez, C., Rubio, A., Sagiv, M.: Taming callbacks for smart contract modularity. In: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2020, vol. 4, pp. 209:1–209:30 (2020)
7. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts (SoK). In: Maffei, M., Ryan, M. (eds.) POST 2017. LNCS, vol. 10204, pp. 164–186. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54455-6_8
8. Bansal, K., Koskinen, E., Tripp, O.: Automatic generation of precise and useful commutativity conditions. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 115–132. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_7
9. Daian, P.: Analysis of the DAO exploit (2016). http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/
10. Grossman, S., et al.: Online detection of effectively callback free objects with applications to smart contracts. PACMPL, 2(POPL) (2018)
11. Liu, M.: Urgent: OUSD was hacked and there has been a loss of funds (2020). https://medium.com/originprotocol/urgent-ousd-has-hacked-and-there-has-been-a-loss-of-funds-7b8c4a7d534c. Accessed 29 Jan 2021
12. Massalin, H.: Superoptimizer - a look at the smallest program. In: Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II), pp. 122–126 (1987)
13. Mehar, M.I., et al.: Understanding a revolutionary and flawed grand experiment in blockchain: the DAO attack. J. Cases Inf. Technol. **21**(1), 19–32 (2019)
14. Nagele, J., Schett, M.A.: Blockchain superoptimizer. In: Proceedings of 29th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR) (2019). https://arxiv.org/abs/2005.05912
15. Tarasov, A.: Millions lost: the top 19 DeFi cryptocurrency hacks of 2020 (2020). https://cryptobriefing.com/50-million-lost-the-top-19-defi-cryptocurrency-hacks-2020/2. Accessed 29 Jan 2021
16. Wang, C., Yang, Z., Kahlon, V., Gupta, A.: Peephole partial order reduction. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 382–396. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_29
17. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger (2019)

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.