Tool Demonstration: FSolidM for Designing Secure Ethereum Smart Contracts

Anastasia Mavridou[1] and Aron Laszka[2(✉)]

[1] Vanderbilt University, Nashville, USA
[2] University of Houston, Houston, USA
alaszka@uh.edu

Abstract. Blockchain-based distributed computing platforms enable the trusted execution of computation—defined in the form of *smart contracts*—without trusted agents. Smart contracts are envisioned to have a variety of applications, ranging from financial to IoT asset tracking. Unfortunately, the development of smart contracts has proven to be extremely error prone. In practice, contracts are riddled with security vulnerabilities comprising a critical issue since bugs are by design non-fixable and contracts may handle financial assets of significant value. To facilitate the development of secure smart contracts, we have created the *FSolidM* framework, which allows developers to define contracts as finite state machines (FSMs) with rigorous and clear semantics. FSolidM provides an easy-to-use graphical editor for specifying FSMs, a code generator for creating Ethereum smart contracts, and a set of plugins that developers may add to their FSMs to enhance security and functionality.

Keywords: Smart contract · Security · Finite state machine
Ethereum · Solidity · Automatic code generation · Design patterns

1 Introduction

In recent years, blockchains have seen wide adoption. For instance, the market capitalization of Bitcoin, the leading blockchain-based cryptocurrency, has grown from $15 billion to more than $100 billion in 2017. The goal of the first generation of blockchains was only to provide cryptocurrencies and payment systems. In contrast, more recent blockchains, such as Ethereum, strive to provide distributed computing platforms [1,2]. Blockchain-based distributed computing platforms enable the trusted execution of general purpose computation, implemented in the form of *smart contracts*, without any trusted parties. Blockchains and smart contracts are envisioned to have a variety of applications, ranging from finance to IoT asset tracking [3]. As a result, they are embraced by an increasing number of organizations and companies, including major IT and financial firms, such as Cisco, IBM, Wells Fargo, and J.P. Morgan [4].

L. Bauer and R. Küsters (Eds.): POST 2018, LNCS 10804, pp. 270–277, 2018.
https://doi.org/10.1007/978-3-319-89722-6_11

However, the development of smart contracts has proven to be extremely error prone in practice. Recently, an automated analysis of a large sample of smart contracts from the Ethereum blockchain found that more than 43% of contracts have security issues [5]. These issues often result in security vulnerabilities, which may be exploited by cyber-criminals to steal cryptocurrencies and other digital assets. For instance, in 2016, $50 million worth of cryptocurrencies were stolen in the infamous "The DAO" attack, which exploited a combination of smart-contract vulnerabilities [6]. In addition to theft, malicious attackers may also be able to cause damage by leading a smart contract into a deadlock, which prevents account holders from spending or withdrawing their own assets.

The prevalence of smart-contract vulnerabilities poses a severe problem in practice due to multiple reasons. First, smart contracts handle assets of significant financial value: at the time of writing, contracts deployed on the Ethereum blockchain together hold more than $6 billion worth of cryptocurrency. Second, it is *by design* impossible to fix bugs in a contract (or change its functionality in any way) once the contract has been deployed. Third, due to the "code is law" principle [7], it is also *by design* impossible to remove a faulty or malicious transaction from the blockchain, which means that it is often impossible to recover from a security incident.[1]

Previous work focused on alleviating security issues in *existing* smart contracts by providing tools for verifying correctness [7] and for identifying common vulnerabilities [5]. In contrast, we take a different approach by developing a framework, called *FSolidM* [9], which helps developers to create smart contracts that are secure by design. The main features of our framework are as follows.

Formal Model: One of the key factors contributing to the prevalence of security issues is the semantic gap between the developers' assumptions about the underlying execution semantics and the actual semantics of smart contracts [5]. To close this semantic gap, FSolidM is based on a simple, formal, finite-state machine (FSM) based model for smart contracts, which we introduced in [9]. The model was designed to support Ethereum smart contracts, but it could easily be extended to other platforms.

Graphical Editor: To further decrease the semantic gap and facilitate development, FSolidM provides an easy-to-use graphical editor that enables developers to design smart contracts as FSMs.

Code Generator: FSolidM provides a tool for translating FSMs into Solidity, the most widely used high-level language for developing Ethereum contracts. Solidity code can be translated into Ethereum Virtual Machine bytecode, which can be deployed and executed on the platform.

Plugins: FSolidM enables extending the functionality of FSM based smart contract using plugins. As part of our framework, we provide a set of plugins that address common security issues and implement common design patterns, which

[1] It is possible to remove a transaction or hard fork the blockchain if the stakeholders reach a consensus; however, this undermines the trustworthiness of the platform [8].

Table 1. Common smart-contract vulnerabilities and design patterns

Type	Common name	FSolidM plugin
Vulnerabilities	Reentrancy [5,10]	Locking
	Transaction ordering [5,10]	Transition counter
Patterns	Time constraint [11]	Timed transitions
	Authorization [11]	Access control

were identified by prior work [5,10,11]. In Table 1, we list these vulnerabilities and patterns with the corresponding plugins.

Open Source: FSolidM is open-source and available online (see Sect. 3).

The advantages of our framework, which helps developers to create secure contracts instead of trying to fix existing ones, are threefold. First, we decrease the semantic gap and eliminate the issues arising from it by providing a formal model and an easy-to-use graphical editor. Second, since the process is rooted in rigorous semantics, our framework may be connected to formal analysis tools [12,13]. Third, the code generator and plugins enable developers to implement smart contracts with minimal amount of error-prone manual coding.

The rest of this paper is organized as follows. In Sect. 2, we present blind auction as a motivating example, which we implement as an FSM-based smart contract. In Sect. 3, we describe our FSolidM tool and its built-in plugins. Finally, in Sect. 4, we offer concluding remarks and outline future work.

2 Defining Smart Contracts as FSMs

Consider as an example a blind auction (similar to the one presented in [14]), in which a bidder does not send her actual bid but only a hash of it (i.e., a blinded bid). A bidder is required to make a deposit—which does not need to be equal to her actual bid—to prevent her from not paying after she has won the auction. A deposit is considered valid if its value is higher than or equal to the actual bid. A blind auction has four main *states*:

1. `AcceptingBlindedBids:` blinded bids and deposits may be submitted;
2. `RevealingBids:` bidders may reveal their bids (i.e., they can send their actual bids and the contract checks if the hash value is the same as the one submitted in the previous state and if they made sufficient deposit);
3. `Finished:` the highest bid wins the auction; bidders can withdraw their deposits except for the winner, who can withdraw only the difference between her deposit and bid;
4. `Canceled:` bidders can retract bids and withdraw their deposits.

Since smart contracts have *states* (e.g., `AcceptingBlindedBids`) and provide functions that allow other entities (e.g., contracts or users) to invoke *actions* that change the current state of a contract, they can be naturally represented as

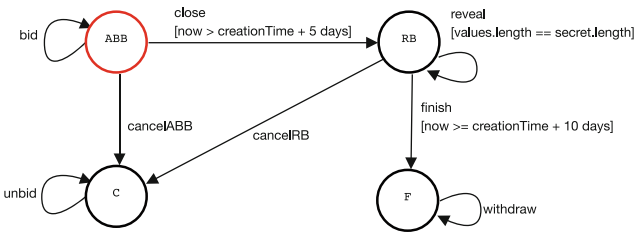

Fig. 1. Example FSM for blinded auctions.

FSMs [15]. An FSM has a finite set of states and a finite set of transitions between these states. A transition forces a contract to take a set of actions if the associated conditions, i.e., the *guards* of the transition, are satisfied. Since such states and transitions have intuitive meaning for developers, representing contracts as FSMs provides an adequate level of abstraction for behavior reasoning.

Figure 1 presents the blind auction example in the form of an FSM. For simplicity, we have abbreviated `AcceptingBlindedBids`, `RevealingBids`, `Finished`, and `Canceled` to `ABB`, `RB`, `F`, and `C`, respectively. `ABB` is the initial state of the FSM. Each transition (e.g., `bid`, `reveal`, `cancel`) is associated to a set of actions that a user can perform during the blind auction. For instance, a bidder can execute the `bid` transition at the `ABB` state to send a blind bid and a deposit value. Similarly, a user can execute the `close` transition, which signals the end of the bidding period, if the associated guard `now >= creationTime + 5 days` evaluates to true. To differentiate transition names from guards, we use square brackets for the latter. A bidder can reveal her bids by executing the `reveal` transition. The `finish` transition signals the completion of the auction, while the `cancelABB` and `cancelRB` transitions signal the cancellation of the auction. Finally, the `unbid` and `withdraw` transitions can be executed by the bidders to withdraw their deposits. For ease of presentation, we omit from Fig. 1 the actions that correspond to each transition. For instance, during the execution of the `withdraw` transition, the following action is performed `amount = pendingReturns[msg.sender]`.

3 The FSolidM Tool

FSolidM is an open-source[2], web-based tool that is built on top of WebGME [16]. FSolidM enables collaboration between multiple users during the development of smart contracts. Changes in FSolidM are committed and versioned, which enables branching, merging, and viewing the history of a contract. We present the FSolidM tool in more detail in [17].

To generate the Solidity code of a smart contract using FSolidM, a user must follow three steps: (1) specify the smart contract in the form of the FSM by using the dedicated graphical editor of FSolidM; (2) specify attributes of the smart

[2] https://github.com/anmavrid/smart-contracts.

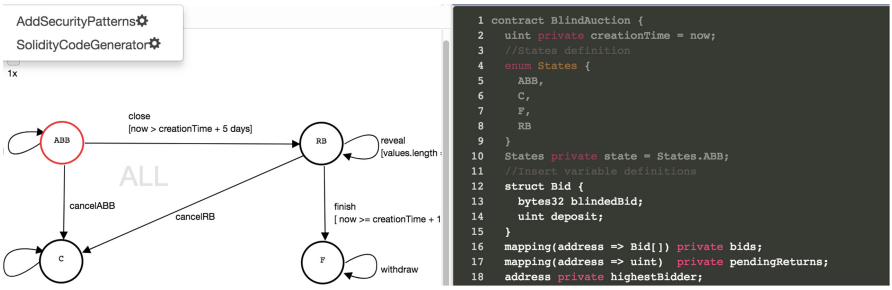

Fig. 2. The FSolidM model and code editors.

contract such as variable definition, statements, etc. in the `Property Editor` or in the dedicated Solidity code editor of FSolidM; (3) optionally apply security patterns and functionality extensions, and finally, generate the Solidity code. Figure 2 shows the graphical and code editors of the tool (for steps 1 and 2) and the list of services (i.e., `AddSecurityPatterns` and `SolidityCodeGenerator` for step 3) that are provided by FSolidM. We have integrated a Solidity parser[3] to check the syntax of the Solidity code that is given as input by the users.

Notice that in Fig. 2, parts of the code shown in the code editor are darker (lines 1–10) than other parts (lines 12–15). The darker lines of code include code that was generated from the FSM model defined in the graphical editor and are locked—cannot be altered in the code editor. The non-dark parts indicate code that was directly specified in the code editor.

FSolidM provides mechanisms for checking if the FSM is correctly specified (e.g., whether an initial state exists or not). FSolidM notifies developers of errors and provides links to the erroneous nodes of the model (e.g., a transition or a guard). Through the `SolidityCodeEditor` service, FSolidM provides an FSM-to-Solidity code generator. Additionally, through the `AddSecurityPatterns` service, FSolidM enables developers to enhance the functionality and security of contracts conveniently by adding plugins to them. Our framework provides four built-in plugins: locking, transition counter, timed transitions, and access control. Plugins can be simply added with a "click," as shown in Fig. 3.

Locking: When an Ethereum contract calls a function of another contract, the caller has to wait for the call to finish. This allows the callee—who may be malicious—to exploit the intermediate state of the caller, e.g., by invoking a function of the caller. This re-entrancy issue is one of the most well-known vulnerabilities, which was also exploited in the infamous "The DAO" attack.

To prevent re-entrancy, we provide a security plugin for locking the smart contract. Locking eliminates re-entrancy vulnerabilities in a "foolproof" manner: functions within the contract cannot be nested within each other in any way.

[3] https://github.com/ConsenSys/solidity-parser.

Fig. 3. Running the `AddSecurityPatterns`.

Transition Counter: If multiple functions calls are invoked around the same time, then the order in which these calls are executed on the Ethereum blockchain may be unpredictable. Hence, when a user invokes a function, she may be unable to predict what the state and the values stored within a contract will be when the function is actually executed. This issue has been referred to as "transaction-ordering dependence" [5] and "unpredictable state" [10], and it can lead to various security vulnerabilities.

We provide a plugin that prevents unpredictable-state vulnerabilities by enforcing a strict ordering on function call executions. The plugin expects a transition number in every function as a parameter and ensures that the number is incremented by one for each function execution. As a result, when a user invokes a function with the next transition number in sequence, she can be sure that the function is executed before any other state changes can take place.

Automatic Timed Transitions: We provide a plugin for implementing time-constraint patterns. We extend our language with timed transitions, which are similar to non-timed transitions, but (1) their guards and assignments do not use input or output data and (2) they include a number specifying transition time.

We implement timed transitions as a modifier that is applied to every function, and which ensures that timed transitions are executed automatically if their time and data guards are satisfied. Writing such modifiers manually could lead to vulnerabilities. For example, a developer might forget to add a modifier to a function, which enables malicious users to invoke functions without the contract progressing to the correct state (e.g., place bids in an auction even though the auction should have already been closed due to a time limit).

Access Control: In many contracts, access to certain transitions (i.e., functions) needs to be controlled and restricted. For example, any user can participate in a typical blind auction by submitting a bid, but only the creator should be able to cancel the auction. To facilitate the enforcement of such constraints, we provide a plugin that (1) manages a list of administrators at runtime (identified by their addresses) and (2) enables developers to forbid non-administrators from accessing certain functions.

4 Conclusion and Future Work

Blockchain-based decentralized computing platforms with smart-contract functionality are envisioned to have a significant technological and economic impact in the future. However, if we are to avoid an equally significant risk of security incidents, we must ensure that smart contracts are secure. To facilitate the development of smart contracts that are secure by design, we created the FSolidM framework, which enables designing contracts as FSMs. Our framework is rooted in rigorous yet clear semantics, and it provides an easy-to-use graphical editor and code generator. We also implemented a set of plugins that developers can use to enhance the security or functionality of their contracts. In the future, we plan to integrate model checkers and compositional verification tools into our framework [12,13] to enable the verification of security and safety properties.

References

1. Underwood, S.: Blockchain beyond bitcoin. Commun. ACM **59**(11), 15–17 (2016)
2. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger. Technical report EIP-150, Ethereum Project - Yellow Paper, April 2014
3. Christidis, K., Devetsikiotis, M.: Blockchains and smart contracts for the internet of things. IEEE Access **4**, 2292–2303 (2016)
4. Vukolić, M.: Rethinking permissioned blockchains. In: Proceedings of ACM Workshop on Blockchain, Cryptocurrencies and Contracts, pp. 3–7. ACM (2017)
5. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 254–269. ACM, October 2016
6. Finley, K.: A $50 million hack just showed that the DAO was all too human, June 2016. Wired https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/
7. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Zanella-Béguelin, S.: Short paper: formal verification of smart contracts. In: Proceedings of 11th ACM Workshop on Programming Languages and Analysis for Security (PLAS), in Conjunction with ACM CCS 2016, pp. 91–96, October 2016
8. Leising, M.: The Ether thief, June 2017. Bloomberg Markets https://www.bloomberg.com/features/2017-the-ether-thief/
9. Mavridou, A., Laszka, A.: Designing secure Ethereum smart contracts: a finite state machine based approach. In: Proceedings of 22nd International Conference on Financial Cryptography and Data Security (FC), February 2018
10. 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
11. Bartoletti, M., Pompianu, L.: An empirical analysis of smart contracts: platforms, applications, and design patterns. In: Brenner, M., Rohloff, K., Bonneau, J., Miller, A., Ryan, P.Y.A., Teague, V., Bracciali, A., Sala, M., Pintore, F., Jakobsson, M. (eds.) FC 2017. LNCS, vol. 10323, pp. 494–509. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70278-0_31

12. Bensalem, S., Bozga, M., Nguyen, T.-H., Sifakis, J.: D-Finder: a tool for compositional deadlock detection and verification. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 614–619. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_45
13. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 334–342. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_22
14. Solidity by example: blind auction. http://solidity.readthedocs.io/en/develop/solidity-by-example.html. Accessed 9 May 2017
15. Solidity specification: common patterns. http://solidity.readthedocs.io/en/develop/common-patterns.html. Accessed 9 May 2017
16. Maróti, M., Kecskés, T., Kereskényi, R., Broll, B., Völgyesi, P., Jurácz, L., Levendovszky, T., Lédeczi, Á.: Next generation (meta) modeling: web-and cloud-based collaborative tool infrastructure. In: Proceedings of MPM@ MoDELS, pp. 41–60 (2014)
17. Mavridou, A., Laszka, A.: Tool demonstration: FSolidM for designing secure Ethereum smart contracts. arXiv:1802.09949 [cs.CR] (2018)