ModCon: A Model-Based Testing Platform for Smart Contracts

Ye Liu
Nanyang Technological University
Singapore
li0003ye@e.ntu.edu.sg

Yi Li
Nanyang Technological University
Singapore
yi_li@ntu.edu.sg

Shang-Wei Lin
Nanyang Technological University
Singapore
shang-wei.lin@ntu.edu.sg

Qiang Yan
WeBank
China
qyan@webank.com

ABSTRACT

Unlike those on public permissionless blockchains, smart contracts on enterprise permissioned blockchains are not limited by resource constraints, and therefore often larger and more complex. Current testing and analysis tools lack support for such contracts, which demonstrate stateful behaviors and require special treatment in quality assurance. In this paper, we present a model-based testing platform, called ModCon, relying on user-specified models to define test oracles, guide test generation, and measure test adequacy. ModCon is Web-based and supports both permissionless and permissioned blockchain platforms. We demonstrate the usage and key features of ModCon on real enterprise smart contract applications.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Smart contract, blockchain, model-based testing

ACM Reference Format:
Ye Liu, Yi Li, Shang-Wei Lin, and Qiang Yan. 2020. ModCon: A Model-Based Testing Platform for Smart Contracts. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3368089.3417939

1 INTRODUCTION

Smart contracts are computer programs that execute on top of blockchains (e.g., Ethereum [26]) to manage large sums of money, carry out transactions of assets, and govern the transfer of digital rights between different parties. Transactions conducted through smart contracts are recorded on blockchains, thus decentralized and immutable, without requiring validation from a central authority. Due to these unique advantages, smart contracts have gained much popularity in recent years. Many believe that this technology has the potential to reshape a number of industries, e.g., banking, insurance, supply chains, and financial services [12].

The existing blockchain networks can be broadly categorized into the permissionless and permissioned blockchains, where the former is open to the public (e.g., Bitcoin [19] and Ethereum [26]) and the latter is only accessible to trusted private groups or individuals (e.g., Hyperledger Fabric [7]). The consortium/federated blockchains (e.g., FISCO BCOS [5] and Azure Blockchain Workbench [3]) sit somewhere in the middle: they are suitable for use between multiple businesses or organizations for performing transactions and exchanging information. One major difference between smart contracts on the permissioned and permissionless blockchains is that the contract execution on permissionless chains is bounded by resource constraints. For example, on Ethereum, one has to pay miners a certain amount of "gas" (cryptocurrency on Ethereum) as the transaction fee to deploy or call contract, which is largely decided by the complexity of the contract (e.g., up to $15 in fees [4]). Therefore, to reduce the gas consumption, smart contracts on permissionless chains are often kept simple, making it unsuitable for implementing enterprise applications with complex business logic.

At the same time, smart contracts have been used to implement many industrial applications of high complexity and production quality on permissioned and consortium blockchains. Unlike the permissionless blockchains, such as Bitcoin and Ethereum mainly used for cryptocurrency exchange (e.g., ERC Token and DeFi applications), the permissioned blockchains aim to create real value. For instance, FISCO BCOS has been successfully adopted in areas such as government and judicial services, supply chain, finance, health care, copyright management, education, transportation, and agriculture [5]. The smart contracts powering these applications are more sophisticated and often demonstrate strong stateful behaviors.

Example. Figure 1 illustrates some usage scenarios of a Credit Management Application (CMA) at WeBank [9], implemented using smart contracts, running on FISCO BCOS consortium blockchain. CMA is used to handle inventory and asset management in supply chain through a blockchain-based credit system, which can facilitate credit transfer among different business owners and help small businesses receive instant financial support securely.

The user first deploys an `AccountController` contract, whose address is then used to instantiate the `CreditController` contract. `AccountController` is in charge of the account creation and

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3417939

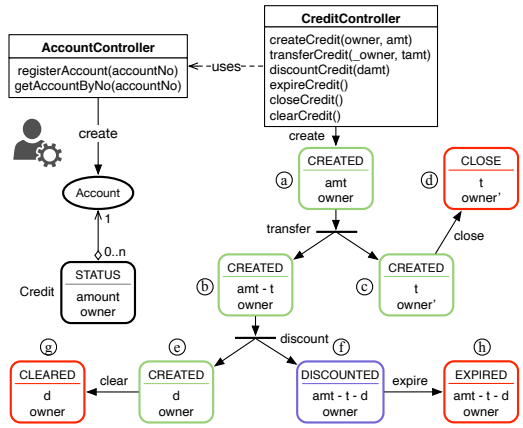


Figure 1: Illustration of a CMA smart contract at WeBank.

management. An account may own `Credit`(s), which are transferable and divisible tokens with stipulated values. The state of a `Credit` is captured by the tuple, (*STATUS*, *amount*, *owner*), whose fields represent the status, value captured, and its ownership, respectively. A `Credit` instance supports credit operations including creation, transfer, discount, expiration, clearance, and closure. Through `CreditController`, one can first create a credit, namely, ⓐ, under the specified `Account`. In this case, a `transfer` operation is executed on ⓐ, thus dividing ⓐ into two new credits, namely, ⓑ and ⓒ. By design, the total value of ⓑ and ⓒ equals to that of ⓐ. Then a `discount` operation is applied on ⓑ, resulting in a newly created credit ⓔ and a discounted credit ⓕ. By design, the total value of ⓔ and ⓕ equals to that of ⓑ, but the status of ⓕ becomes "DISCOUNTED". To complete the life cycle of a credit, one may apply either the `close`, `clear`, or `expire` operation, bringing the credit into the "CLOSED" (e.g., ⓓ), "CLEARED" (e.g., ⓖ), or "EXPIRED" (e.g., ⓗ) state, respectively. Once a credit is in "CLOSED /CLEARED/EXPIRED", it should no longer accept further operation.

Existing testing and analysis tools target Ethereum smart contracts and mainly focus on their security issues. Such tools do not work well on this example for the following reasons. **(1) Lack understanding of system behaviors**. The different states of a credit instance is implemented with special encoding. For example, the *STATUS* field is encoded as bit-vectors for performance considerations. It is unclear how to interpret system states and behaviors at these states without this knowledge. **(2) Absence of oracle**. Existing tools may rely on implicit security properties (e.g., underflow/overflow and exceptions) as oracle, which is absent when the functional correctness is concerned. The expected system behavior (e.g., "EXPIRED" is terminal) is not known prior and should be provided by the contract designer. **(3) Missing measurement of test adequacy**. The traditional coverage criteria used by existing tools, such as branch and path coverage, are not good measurement of test adequacy for this example. Covering every single path of the contract program does not equal exercising all system states and state transitions. It is challenging to navigate through all system behaviors without proper adequacy measurements.

ModCon. To address these challenges, we propose ModCon, a model-based testing platform for smart contracts. ModCon targets enterprise smart contract applications written in Solidity [21] from

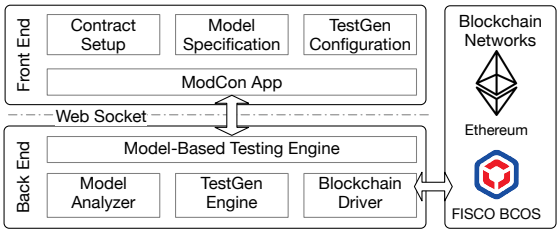


Figure 2: Architecture of ModCon.

permissioned/consortium blockchains such as FISCO BCOS, but is also compatible with Ethereum.

ModCon allows users to specify system models and define test oracles, which are then used to guide the test generation and execution. The key features of ModCon include the following.

- **Test-Model Specification**. ModCon allows users to provide a test model for the target smart contract. The model is used to specify the state definitions, expected transition relations, pre/post conditions to be satisfied for each transition, invariants, and the mapping from the model to the contract code.
- **Customized Test Generation**. With the test model given, users can further customize the testing process by choosing from different coverage strategies and test prioritization options. ModCon then generates tests with the goal of exercising as many system behaviors as possible while prioritizing on cases of particular interests. Any violation of the specified oracle is recorded and reported to users.
- **Web-Based Interface**. ModCon has a Web-based interface, providing easy access to all the testing capabilities and customization options. Source code and a video demonstrating the usage of ModCon are available at https://sites.google.com/view/modcon.

2 MODCON OVERVIEW

In this section, we describe the architecture of ModCon and demonstrate its user interface. As shown in Fig. 2, ModCon consists of a web-based front end (implemented as a Vue.js [8] application) and a server-side back end (implemented on top of the Node.js JavaScript runtime [6]). The front-end accepts two inputs from users: the target smart contracts and the test-model specifications to drive the model-based testing process. The front-end allows users to specify coverage strategies and configure test generation priority, and the test execution progress can be monitored on-the-fly. The back-end communicates with the front-end through the WebSocket. On the back-end, the model-based testing engine is in charge of smart contract compilation/deployment, model specification analysis, and the customized model-based testing tasks as per users' requests.

2.1 User Interface

The user interface of ModCon mainly supports three tasks, namely, contract setup, model specification, and testing controls.

Contract Setup. First, users are to upload all relevant smart contract source files, which are then automatically compiled and deployed onto the blockchain network. Once the contracts are successfully deployed, users can directly interact with them by sending transactions, and the transaction receipts are displayed on the result pane below. For example, as shown in Fig. 3, seven contracts related to the CMA application (i.e., `Account`, `AccountController`,

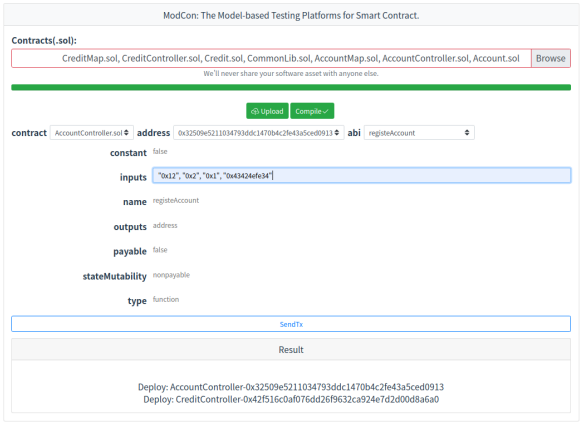


Figure 3: Smart contract deployment and setup.

```
"id": "CMA#1",
"main": "CreditController",
"contracts": {
  "CreditController": {"address": "0x00", "name": "CreditController"}
},
"actions": {
  "create": {"CreditController": ["createCredit"]},
  "discount": {"CreditController": ["discountCredit"]},
  "transfer": {"CreditController": ["transferCredit"]},
  "expire": {"CreditController": ["expireCredit"]},
  "clear": {"CreditController": ["clearCredit"]},
  "close": {"CreditController": ["closeCredit"]}
},
"states": [
  { "name": "CREATED", "type": "regular", "Predicate": "status[0] == CREATED" },
  ...
],
"transitions": [
  { "from": "INITIAL", "to": "CREATED", "action": "create" },...
]
```

Figure 4: User-configured model specification.

`Credit`, `CreditController`, ...) had been uploaded to ModCon. `AccountedController` and `CreditController` were deployed at addresses, "`0x325...913`" and "`0x42f...6a0`", respectively. One transaction, calling the `registerAccount` function, was sent to `AccountController` to create an account which would be used to hold credit instances. The target contracts' information, including the ABIs and deployment details, are cached and will be used for test case generation in a later stage.

Model Specification. Figure 4 shows an abridged test-model specification for the CMA application, which user can customize for his/her applications. The "`id`" and "`main`" fields indicate the model identifier and the entry contract, respectively. The "`contracts`" field lists all relevant contract dependencies required in the test model. The "`states`" and "`transitions`" fields jointly define a state machine model for the target application. The "`actions`" field establish a mapping between functions from the contract implementation and the actions that can be taken to perform state transitions.

TestGen Configuration. The test-model specification (i.e., Fig. 4) provided by users is visualized as a state machine diagram shown in Fig. 5. Users may further customize the test generation process by choosing from the three coverage strategies: (1) *cover states*, aiming to cover every states, (2) *cover transitions*, aiming to cover every transitions, and (3) *cover transitions (loop)*, aiming to cover every transitions including loops. Based on experiments, covering loop transitions may increase testing costs without covering new states, but it can help discover corner cases and verify the integrity of the test-model. In addition, users may prioritize the test generation leaning towards specific states or transitions. As shows in Fig. 5,

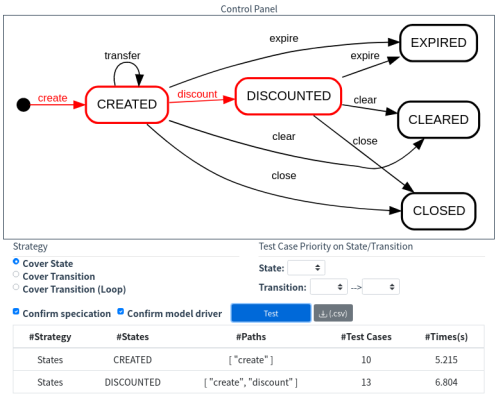


Figure 5: Test generation control panel.

the "cover states" strategy is selected and the states `CREATED` and `DISCOUNTED` are covered by 10 and 13 test cases, respectively.

2.2 Back-End Implementation

The model-based testing engine consists of three parts: i.e., model analyzer, test generation engine, and blockchain driver.

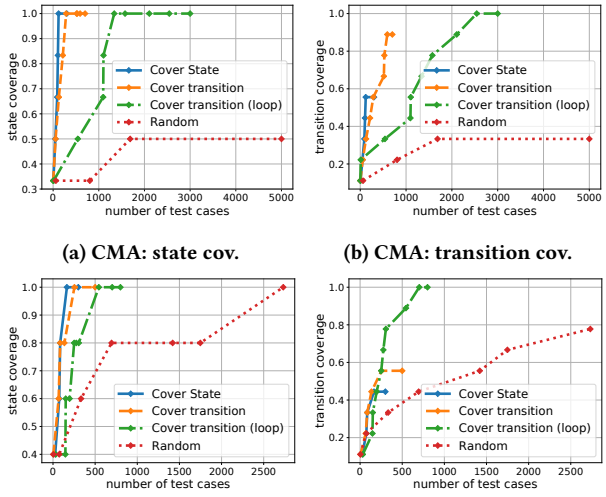
Model Analyzer. The model analyzer reads the model specification from the front-end and automatically translates it into a test driver written in JavaScript. The test driver stipulates how tests should be generated and executed, which is then displayed in the front-end client for users' confirmation and customization. For instance, users may insert additional test oracles in the form of pre/post conditions and assertions.

TestGen Engine. The test generation (TestGen) engine receives testing requests and collects the test-model related information from the front-end, which includes the confirmed test driver, the coverage strategies, and the test generation priorities. The engine first computes all logical transition paths following the specific coverage strategies and goals using graph searching algorithms. For example, to reach the `CLEARED` state of CMA shown in Fig. 5, the logical transition paths for different strategies are listed below.

- Cover states: INITIAL → CREATED → CLEARED.
- Cover transitions: INITIAL → CREATED → CLEARED; INITIAL → CREATED → DISCOUNTED → CLEARED.
- Cover transitions (loop): INITIAL → CREATED → CREATED → CLEARED; INITIAL → CREATED → CREATED → DISCOUNTED → CLEARED.

The TestGen engine ranks these logical transition paths based on the order defined by the test case priorities, and then generates concrete test cases (with concrete input values and environment settings) corresponding to each logical transition path. The generation of concrete input values adopts standard techniques, such as the mutation-based method in ContraMaster [24, 25], with seed pools for different input types. Built upon the blockchain driver, the TestGen engine sends these concrete test cases to blockchain platforms for execution and monitors the execution status at the same time. The engine keeps generating test cases for execution until the maximum time budget or failure limit is reached. During test execution, the engine reports the testing results back to the front-end client, which displays the current progress in real-time.

Blockchain Driver. The blockchain driver directly interacts with the blockchain networks for contract deployment and establishes



(a) CMA: state cov. (b) CMA: transition cov.

(c) `BlindAuction`: state cov. (d) `BlindAuction`: transition cov.

Figure 6: State and transition coverage achieved per test for CMA and `BlindAuction`.

a transaction interface with the networks. Currently, ModCon supports two blockchain platforms, namely, Ethereum and FISCO BCOS. It can easily be extended to other blockchain platforms.

3 EVALUATION

In this section, we evaluate ModCon on the CMA smart contract application from WeBank and the `BlindAuction` contract used by FSolidM [17], a state machine based smart contract code generator.

We manually constructed their model specifications with the help from the contract developers and the related documentation. The experiments were conducted on a desktop computer with Ubuntu 18.10 OS, an Intel Core i5 2.50 GHz processor and 8GB RAM. All cases were evaluated on the FISCO BCOS blockchain.

Figure 6 shows the evaluation results. The vertical and horizontal axes represent the state/transition coverage and the number of test cases, respectively. We examined aforementioned three coverage strategies and compared the results of ModCon with random testing. Among these strategies, the results show that the cover state strategy first reaches all states of both CMA and `BlindAuction`, while the strategy to cover transition including loops has the potential to reach all states and explore more transitions at the cost of more test cases. All of the three proposed strategies achieve much higher state and transition coverage than random testing, which shows that random testing is not suitable to deal with enterprise smart contract applications. Random testing achieves lower state and transition coverage in CMA than those in `BlindAuction`, because the former is of higher complexity in its business logic and state encoding than the latter. For example, CMA uses a bit-vector of more than 11 bits as its function input or to encode the *STATUS*, and blindly enumerating bit-vector values is extremely inefficient.

In our experiments, ModCon was able to reach all states and transitions for each case within about 500 test cases. This is mainly because of the guidance from the test-model, which makes ModCon effective on enterprise smart contract applications such as CMA. Additionally, with the test-model specification, ModCon allows users to define test oracles in the generated test driver. For example, the specification of CMA requires `CLOSED`, `CLEARED`, and `EXPIRED` to be final states, which means no transition shall be made once the system falls into one of the three states. We insert this specification as a test oracle into the test driver and discovered violations against it in the original implementation of CMA. The transitions between `EXPIRED`, `CLOSED`, and `CLEARED` were possible due to an implementation error. We reported this error to the CMA developer team from WeBank, and they confirmed it to be a real bug. The demonstration video of ModCon, along with more cases and experiment results, can be accessed at: https://sites.google.com/view/modcon.

4 RELATED WORK

Most of the existing testing and analysis tools focus on the security issues of Ethereum smart contracts. Oyente [2, 16] is one of the first static analyzer detecting security vulnerabilities in smart contracts based on symbolic execution. It searches for violations of predefined security properties without actually executing the contract program. Other notable static security analysis tools include Zeus [14], Mythril [1], sCompile [11], and Securify [22]. In contrast, the dynamic tools instrument either the contract code or the Ethereum Virtual Machine (EVM) and observe anomalies during runtime execution. ContractFuzzer [13] is the earliest dynamic fuzz testing tool aiming a number of common vulnerability types, including the reentrancy, exception disorder, block dependency, etc. Other fuzzing tools follow similar principles: e.g., Reguard [15], ContraMaster [24, 25], and sFuzz [20]. These tools are not designed for testing functional correctness, and as mentioned in Sec. 1, they are not suitable for enterprise smart contract applications either.

There are several recent works on the functional correctness of Ethereum smart contracts. VeriSol [10] relies on formal verification to check the semantic conformance between a contract implementation and its workflow policy. The policy is provided by users, describing the high-level workflow of the application in a style similar to our model specifications. FSolidM [17] and VeriSolid [18] both aim to facilitate the creation of correct-by-design contracts, with emphases on the security and functional aspects, respectively, where a finite state machine is used as the contract specification to capture the expected system behaviors. ModCon is based on the idea of model-based testing [23], which uses an explicit abstract model of the target contract to automatically derive tests. It serves as a complement to other static validation/construction techniques in providing more flexible and accurate quality assurance solutions.

5 CONCLUSION

In this paper, we described the architecture of ModCon, its user interface, and prominent features. We also demonstrated the effectiveness of it on real smart contract applications from WeBank. The model-based testing capability of ModCon enables it to generate higher-quality test cases for enterprise smart contracts from permissioned and consortium blockchains.

ACKNOWLEDGMENTS

This research is partly supported by the Joint NTU-WeBank Research Centre of Eco-Intelligent Applications (THEIA), Nanyang Technological University, Singapore (NWJ-2019-003).

REFERENCES

[1] 2019. Mythril. https://github.com/ConsenSys/mythril. A Security Analysis Tool for EVM Bytecode.

[2] 2019. Oyente. https://github.com/melonproject/oyente. An Analysis Tool for Smart Contracts.

[3] 2020. Azure Blockchain Workbench. https://azure.microsoft.com/en-us/features/blockchain-workbench/.

[4] 2020. Ethereum Transaction Fees Fall by 75% as Congestion Eases. https://cointelegraph.com/news/ethereum-transaction-fees-fall-by-75-as-congestion-eases.

[5] 2020. FISCO BCOS. https://fisco-bcos.org/.

[6] 2020. Node.js. https://nodejs.org/.

[7] 2020. Open, Proven, Enterprise-Grade DLT. https://www.hyperledger.org/wp-content/uploads/2020/03/hyperledger_fabric_whitepaper.pdf.

[8] 2020. The Progressive JavaScript Framework. https://vuejs.org/.

[9] 2020. WeBank (China). https://en.wikipedia.org/wiki/WeBank_(China).

[10] Cody Born, Immad Naseer, and Kostas Ferles. 2020. Formal Verification of Workflow Policies for Smart Contracts in Azure Blockchain. In *Verified Software. Theories, Tools, and Experiments: 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13–14, 2019, Revised Selected Papers*, Vol. 12031. Springer Nature, 87.

[11] Jialiang Chang, Bo Gao, Hao Xiao, Jun Sun, and Zijiang Yang. 2019. sCompile: Critical Path Identification and Analysis for Smart Contracts. In *International Conference on Formal Engineering Methods*. 286–304.

[12] Marco Iansiti and Karim R Lakhani. 2017. The Truth about Blockchain. *Harvard Business Review* (2017).

[13] Bo Jiang, Ye Liu, and WK Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 259–269.

[14] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *NDSS*.

[15] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. Reguard: Finding Reentrancy Bugs in Smart Contracts. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 65–68.

[16] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM, 254–269.

[17] Anastasia Mavridou and Aron Laszka. 2018. Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach. In *International Conference on Financial Cryptography and Data Security*. Springer, 523–540.

[18] Anastasia Mavridou, Aron Laszka, Emmanouela Stachtiari, and Abhishek Dubey. 2019. VeriSolid: Correct-by-Design Smart Contracts for Ethereum. (jan 2019). arXiv:1901.01292 http://arxiv.org/abs/1901.01292

[19] Satoshi Nakamoto et al. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. (2008).

[20] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts. *arXiv preprint arXiv:2004.08563* (2020).

[21] Solidity 2018. Solidity. https://solidity.readthedocs.io/en/v0.5.1/.

[22] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 67–82.

[23] Mark Utting, Alexander Pretschner, and Bruno Legeard. 2012. A Taxonomy of Model-Based Testing Approaches. *Software testing, verification and reliability* 22, 5 (2012), 297–312.

[24] Haijun Wang, Yi Li, Shang-Wei Lin, Cyrille Artho, Lei Ma, and Yang Liu. 2019. Oracle-Supported Dynamic Exploit Generation for Smart Contracts. *arXiv preprint arXiv:1909.06605* (2019).

[25] Haijun Wang, Yi Li, Shang-Wei Lin, Lei Ma, and Yang Liu. 2019. Vultron: Catching Vulnerable Smart Contracts Once and for All. In *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results*. IEEE Press, 1–4.

[26] Gavin Wood. 2014. Ethereum: A Secure Decentralised Generalised Transaction Ledger. *Ethereum project yellow paper* 151 (2014), 1–32.