StateFuzz: Learned Smart Contract Fuzzing on State Transition

Program: Graduate
Jackson Rusch (ruschjp, jackson.p.rusch@vanderbilt.edu)

Dec 5, 2023

Overview. This work aims to create a novel learned oracle detector for fuzzing of Web3 smart contracts through the use of machine learning on the positive and unlabeled (PU) data public on the block chain. Fuzzing has been used extensively to find bugs in various code systems in the past, requiring a system to generate valid inputs / actions to blocks of code and an oracle to determine whether the input invoked a bug in the code. In many systems, this oracle can be easily defined (ex: computer crash). However, for smart contracts there is a large list of potential bugs and corresponding rule defined oracles, making oracle definition non-trivial. Because the blockchain is decentralized, we can leverage the data of users interact with smart contracts in order to discover inputs that are untested and bug prone. By creating a machine learning model that can learn the job of an oracle under the context of smart contracts, we can provide a scalable solution to fuzzing different types of bugs. We will test this method through various evaluation pipelines and evaluate the efficacy of this proposed approach.

Intellectual Merit. Fuzzing has been a well-explored area of smart contracts due to the high risk of poorly tested smart contracts. Fuzzing can be split into three main processes: first an input must be generated, second the input must be run through the program, and finally the outcome of the program must be compared with an oracle to determine if an error occurred. The first two aspects of fuzzing have been researched in the context of smart contracts. Specifically, various methods have been proposed to optimize for representative inputs due to the depth of smart contracts as well as optimize the runtime and cost of running fuzz inputs (through the use of testnets). However, current research has only used predefined oracles (such as markers for a reentrancy vulnerability) that do not scale. The novelty of this research is that oracles are modeled through a learned process, which allows oracles to scale with the use of smart contracts. The hypothesis of this research is that the current amount of publicly accessible data will already allow for better and more diverse oracles than can be enumerated by humans, especially for smart contract bugs that are less defined (such as ones that a novice blockchain programmer would make).

Broader Impact. Smart contracts have high risk due to their immutability and access cryptocurrency assets. From the perspective of software engineers, this perceived risk is an additional barrier to entry. Thus, by creating a system that can identify a wide variety of vulnerabilities and bugs in smart contracts, this barrier is lowered and software engineers can more easily learn to program within the Web3 context. The broader impact of this work is two fold as it prevents smart contract bugs from being deployed and allows for learning opportunities for software engineers to see what kind of inputs can lead to bugs and vulnerabilities (preventing future potential bugs).

1 Introduction, Background, and Related Work

1.1 Web3 Smart Contracts

Smart contracts are immutable programs deployed to the Ethereum blockchain. These contracts are effectively formatted as classes that are deployed on a Web3 distributed system. Smart contracts have an associated wallet, state variables, constructors, and functions. Users can interact with smart contracts through a transaction where they call an individual function. During a transaction, the user provides the function they desire to call, the inputs to that function, and gas (money to pay the blockchain to compute to do their work). Transactions may succeed or fail (similar to throwing an exception). Upon failure, contract state is always reverted to the original state before the transaction started. All of the data corresponding to a transaction is stored on the blockchain.

1.2 Dangers of Vulnerable Smart Contracts

Web3 is primarily used as a distributed financial system. Smart contracts act as the brokers within this financial system, where unchangable bytecode can handle transfers of money, ownership of items, etc. In 2022, it was estimated that approximately $30 billion in assets were under Web3 management [5]. Because of the large payout and infancy of Web3 applications, smart contracts become targets for hacking. There are a number of cases where smart contract vulnerabilities have lead to major losses of assets:

1. In 2016, the DAO smart contract was hacked through a reentrancy attack, causing losses of 3.6 million ETH which was worth $150 million at the time [1] or over $1 billion in 2023 [6]. The reentrancy attack works by requesting repeated calls to a smart contract in a callback function. For example, given the function to transfer money from the smart contract to a private wallet, a software engineer can design the function to first check if that wallet has the funds within the contract, then send the money, then subtract the amount from the state variables. However, a callback function can be attached upon receiving money. An adversarial written callback can transact the transfer function again, creating an infinite loop because the subtraction is never performed.
2. In 2017, the Pairity Multisig Wallet smart contract was hacked through a delegatecall change of ownership attack stealing approximately $30 million in ETH [2]. Often, smart contracts have state variables to track the ownership of the smart contract. As such, some functions are restricted to only be called by the owners (or deployers) of the contract. This is a way increase maintainability and security of the contract. In this hack, a delegatecall function was used to execute arbitrary function calls on a smart contract to change the owner state variable. This allowed the hacker to add their address to the list of owners and siphon money from the wallet contract.

Overall, it has been estimated that vulnerabilities in smart contract have lead to a loss of $4.75 billion [4]. It is thus important to understand why there are so many vulnerabilities in smart contracts and how to prevent them from occuring.

1.3 Existing Problems within Smart Contracts

Due to the nature of distributed blockchain systems and the infancy of Web3 infrastructure, there are a number of current issues within Web3 that make software development difficult. For example, the immutability of smart contracts makes it impossible to change deployed smart contracts, and developers need to ensure that their systems are built with fail safes in mind. Additionally, the cost and time associated with computing transactions on real networks makes it difficult to test contracts in the field. The creation of testnets (which provide lower validation requirements for correct computation) allow for cheaper testing of smart contracts pre-deployment. Despite these two solutions, survey research of smart contract software engineers showed that 72.4% believed testing smart contracts was harder than normal languages and 69.4% believed that testing was difficult because it was hard to consider all edge cases and scenarios in unit testing [17]. In order to increase confidence in deployments and lower the number of vulnerabilities within smart contracts, there is a requirement to make edge case discovery easier for software engineers.

1.4 Fuzzing

In 1990, the software engineering strategy of fuzzing was developed as a way to measure the reliability of Unix systems [10]. Fuzzing was created as a system to test a wide variety of randomly generated inputs inputs on a program to check with random use how often the program failed. There are three major aspects to fuzzing [3]:

1. Poet: The poet is responsible for generating inputs to run into the program. Ideally, the oracle generates inputs that have the ability to discover a wide variety of problems efficiently.
2. Courier: The courier is responsible for providing the input to the program and collecting the output. Essentially, the courier runs/simulates the program with the given input.
3. Oracle: The oracle determines the validity of the given output, often defined by a set of conditions or markers that must be met. For example, a common marker for an oracle is that the program does not crash (exit unexpectedly).

Nowadays, fuzzing is often used as a system to ensure software reliability in convoluted systems, allowing for the discovery of edge cases difficult for software engineers to come up with.

1.5 Fuzzing Smart Contracts

Because the major issue as identified before for smart contracts was the difficulty to consider all possible edge cases in testing, fuzzing presents a natural solution to this problem. Specifically, before deployment of a smart contract, fuzzing analysis can be run to identify any issues that were not already tested for, decreasing the chance of deploying vulnerabilities and increasing confidence within the software engineers. In fact, significant research has gone into understanding how to most efficiently and accurately build a smart contract fuzzing system. [9] developed ContractFuzzer which used open source smart contract code to create a more efficient fuzzing poet and validated this through the use of seven defined oracles and past hacks such as the DAO attack. Echidna [7] incorporated static analysis information to create a more efficient poet especially in deep smart contracts. This research used code reachability as their oracle. Harvey [15] uses a similar method but also optimizes the poet for code reachability in as few transactions as possible. sFuzz [11] created a more efficient poet for deep smart contracts using eight vulnerability oracles. ILF [8] approached input generation through a machine learning lens and validated their poet using 6 predefined oracles. As demonstrated, these current state of research into fuzzing smart contracts has primarily focused on efficiency of the poet - to generate inputs that more efficiently expose vulnerabilities of a smart contract. However, these works all defined a subset of oracles that do not represent the space of all possible bugs in smart contracts. The oracles defined were often the most common/severe vulnerabilities (for example reentrancy). This strategy is ineffective at scale especially in the context of novice smart contract software engineers, as they can make simple mistakes that lead to uncommon vulnerabilities.

Thus, the main research questions of this work are:

- How can we develop a learned fuzzing oracle for the use in smart contracts leveraging the data on the blockchain?
- In what contexts would this fuzzing system provide benefits to smart contract software engineers?

2 Proposed Research

This project will develop a learned smart contract fuzzing oracle through the use of collected positive and unlabeled (PU) data. A modified Observer Generative Adversarial Network (observer GAN) [16] model will be used to understand whether a state transition in a smart contract is valid or invalid - which will be used as the marker for a vulnerability. The rest of this section will explain the plan for major aspects of the proposed research.

2.1 Data Collection

Transactions of smart contracts are publicly available on tools such as etherscan.io. The transaction inputs and outputs are available directly for each transaction made. However, this does not give access to the state transition of the smart contract itself. Thus, in order to collect the state transition on the positive transactions (those that were run in the real world), we must first collect the code for the deployed smart contract (most is available on etherscan), add getters to access state variables, compile it locally, and replay the transactions (in order based on the block number) to collect the state transitions. To collect unlabeled data, we will run random inputs through the collected smart contracts and collect the state transition of each transaction. This data is unlabeled because it can potentially degenerate the state of the smart contract (representing an untested input) whereas we assume that transactions that have taken place on the blockchain do not degenerate the state of the smart contract.

2.2 Development of the Oracle Model

Using the positive and unlabeled data collected, we will train an observer GAN model to distinguish between state transitions that do and do not degenerate contract state. Observer GANs work by training three models concurrently. The Observer will learn to distinguish the difference between positive and generated inputs. The Discriminator will learn the difference between unlabeled and generated inputs. The Generator will learn to predict samples the discriminator believes are real unlabeled samples and the observer believes are not positive samples (thus learning to generate valid but negative samples). It is important to note that observer GAN models are primarily used within the context of images which have much higher dimensionality compared to our state transition. This leads to a greater risk of mode collapse within the generator, where the generator maximizes its objective by generating a small set of values (with low diversity). Previous research has looked into preventing mode collapse within vanilla GANs, for example with the use of WGANs [14], diversity penalties [12].

In addition to the state transition, in the full work we plan add a latent representation smart contract function using LLMs to the observer and discriminator parts of the observer GAN to encode context on output expectation. This allows the model to gain an understanding of what a function is supposed to do which is essential to understanding whether or not the transaction degenerated the contract state. This will allow a single model to be used over multiple contracts, which can gain a more general understanding of bugs and require less data from a single contract to perform effectively.

2.3 StateFuzz

Given the oracle model, we will package it into a fully functional fuzzing solution with a poet to generate random inputs and a courier to locally run transactions on an Ethereum Virtual Machine (EVM). It is important to note that the PU data problem that is training the observer GAN with unlabeled data created from random inputs is effectively a reformed fuzzing problem. However, just running the observer model on a test set of unlabeled data may be inferior to using different poets. Additionally, with the addition of the smart contract latent representation using LLMs, it will be important to assess StateFuzz on contracts it has not seen before.

3 Proposed Experiments and Metrics

We plan to run multiple experiments in order of importance, looking for certain criteria for success on each experiment. In general, these experiments add complexity, where initial experiments are toy problems presenting an initial proof-of-concepts and later experiments are the full research experiments more closely addressing the problem of vulnerabilities in smart contracts.

3.1 Observer GAN Proof of Concept

Given that Observer GANs are commonly used for images, we must first validate that these models are a usable strategy at discovering potentially vulnerable inputs to smart contracts. We will create a toy dataset

comprising of a basic smart contract with a single vulnerability. We will then generate positive data by running inputs into the smart contract that do not exploit the vulnerability, simulating positive blockchain data of this smart contract. Additionally, we will run random inputs into the smart contract to simulate unlabeled data that would be gathered in our process. We will evaulate the observer of a training run through a separate test set of unlabled data (annotated with whether or not the vulnerability was actually triggered), using the Receiver Operator Characteristic (ROC) curve of the observer as well as the Precision Recall (PR) curve of the observer. Additionally, we will inspect the outputs from the generator to ensure that the results appear to be well-calibrated (samples that appear similar to the negative samples from the unlabeled data). Unfortunately, common metrics for generator evaluation in GANs is not applicable to this use case because they require feature extraction from images. During this experiment, we will try out different methods at combating mode collapse in the state transition data format.

3.2 Observer GAN Real Data

Next, we will validate that our proof of concept translates to real contracts that are deployed on the blockchain. The dataset from the VeriSmart paper [13], concatenates the CVE, Zeus, SmartBugs, and their own annotation. This is a comprehensive dataset comprising of 616 smart contracts with annotations of vulnerabilities localized per line and categorized for each smart contract (with some smart contracts being free of vulnerabilities). We will start with this dataset and scrape etherscan for the relevant transaction data for each smart contract. Then, we will replay the transactions on a local testnet to generate the state transition dataset of positive samples. Finally, we will generate unlabeled samples using random inputs to transactions on (starting with an uninitialized smart contract as well as the state at specific blocks within the positive data). Using this data, we will train the same observer GAN model as the initial proof of concept on each contract (with potentially some changes to hyperparameters). This model will then be evaluated given the labels provided by the VeriSmart dataset. Specifically, we will set a ground truth validity of each transaction to be defined as whether or not the transaction called the line of code with a vulnerability. The observer will be evaluated using the ROC and PR of predicted vulnerable transactions versus labeled vulnerable transactions. It is important to note that though the vulnerable line of code may be run, this may not actually trigger the vulnerability (meaning our ground truth function is biased towards vulnerabilities). Thus, we will also validate this general labeling method by providing the correct ground truth logic for a small subset of the smart contracts, which we expect to show the same performance as the biased ground truth labeling.

After validating that the observer GAN works on real data, we will add the latent representations of the contract function using a CodeBERT model fine-tuned for smart contracts. Using this extra data, one observer GAN model will be trained and evaluated in a similar method to the above. However, instead of generating new unlabeled data for each contract to use as a test set, we will instead leave out entire smart contracts to see if the observer GAN can generalize across smart contract (or if that is too difficult for the model we will look to find how little data can be used to seed the observer GAN to get acceptable performance).

3.3 Baselines

With the real data observer GAN test, we will compare the class-wise recall and precision of smart contract vulnerabilities against baselines from other research (such as ContractFuzzer, Echidna, etc.). We will specifically be looking for what types of vulnerabilities StateFuzz struggles with as well as what it excels at that is often not a human-defined oracle.

3.4 Future Directions

Depending on the efficacy of this method, there are some potential avenues for future directions of work. If the method appears successful, we can employ a user study to understand how this can be useful across different skill levels of Web3 software engineers. Additionally, we can do an exploratory experiment on smart contracts with no current labels and use the observer GAN as a way to more efficiently label smart contract vulnerabilities. If the method has shortcomings, we can look into adding more novel research that is coming

out about GANs and consider changing models. A system could also be designed that leverages the flexibility of machine learned models as well as the explicitness of human defined oracles, such as training based on the blockchain data when to turn a human defined oracle on or off.

4 Preliminary Results

4.1 Observer GAN Proof of Concept

A toy dataset was created under accordance with the experiment explained above representing a Wallet. There was a single state variable "balance". Two functions (deposit and withdraw) each with a single parameter "amount". Negative amounts sent to either function would fail the transaction. An unsigned integer overflow and underflow bug was put into the contract, where above a set amount and below 0, the balance would wrap around (this is a common error in smart contracts). Positive data was simulated by randomly choosing to deposit or withdraw and randomly choosing an amount that would not cause an overflow / underflow. Unlabeled data was simulated by randomly choosing to deposit or withdraw and randomly choosing an amount with no restrictions.

Due to the nuances of the problem, initial experiments were run to understand how the observer GAN can be best suited to the problem. First, the basic observer GAN was created which was occasionally leading to mode collapse, making the observer succeed on certain cases and fail on others. Next, a WGAN model was attempted though this generally led to worse performance over (possibly due to discrepancies in learning rates between the models in the WGAN format or due to the low dimensionality). Upon further experimentation with the basic observer GAN, we found that it generally worked better to do staggered resets for all models, not just the observer which was mentioned in the original paper. This makes sense because improved information is passed between the models over time, while the models are reset to prevent getting stuck in local minima. Though the observer generally worked better using this method, the generator started producing less realistic results. We wanted to try the diversity loss mentioned by other papers as a way to improve the observer without degrading the generator, but there was no obvious way to translate the image based methods they used to the transaction domain (though this could be something to explore in the future).

Given the initial experiments, the model we decided to further explore was the vanilla observer GAN with staggered resets. Next, it was necessary to understand how the ROC and PR evaluation pipelines were effected by differences in the data. Specifically, the number of positive samples, the number of unlabeled samples, and the percent of unlabeled samples that are positive. We expected that increasing the number of positive and unlabeled samples would improve the models as they would be exposed to more diverse data. Additionally, we expected that the performance of the observer would degrade as the number of negative unlabeled samples decreases (as bug transactions are harder to distinguish from the positive transactions), so we wanted to ensure that the dropoff in performance is reasonable. We trained and evaluated 5 models with different parameter settings ranging from 10^2 to 10^6 number of positive and negative samples as well as unlabeled positive percentage of 0.75, 0.9, and 0.99. It is important to note that due to the data imbalance, the AUC ROC metric for a random model is 0.5 whereas the AP metric for a random model is 0.25, 0.1, and 0.01 for the unlabeled positive percentage of 0.75, 0.9, and 0.99, respectively. 1 shows the results from this experiment. The major takeaways was the AUC ROC metrics was generally unchanged whereas the AP of bugged transactions drastically decreased as the percent of negative data in the unlabeled data decreased. At 0.75 the performance of the observer was relatively good, at 0.9 the performance was still better than random but there was an increase in the variance of the performance, and at 0.99 the model was effectively unusable. The results show that the model shows promise for this problem, but changes must be made so that it can work under less learnable conditions (for example including a diversity loss on the generator).

4.2 Observer GAN Real Data

Due to the experimentation required to understand the efficacy of the observer GAN under this context, little time was left to fully search into this experiment. To show the the beginnings of the work on this experimentation, the etherscan scraper was created to understand how the creation of the dataset would

Variable	Value
transaction_hash	0x78b8fba27470f2a1828e688fef8601860e90c6230fb930a4525d3f204bcbb4f6
block	16752059
from_address	0x20baCb982b65eeCCd6eB95ff466B264eC10DEB63
to_address	0xc5d105e63711398af9bbff092d4b6769c82f793d
function	”transfer(address _to, uint256 _value)”
function_hash	0xa9059cbb
transaction_return.success	0.0
function_parameters._to.address	0xE4DaF5B77Bf9416B600aC6aDec016Bd8166578ad
function_parameters._value.uint256	18000000000000000000
function_parameters._receivers.address[]	None

Table 1: Example transaction data scraped from etherscan for the smart contract deployed to address 0xc5d105e63711398af9bbff092d4b6769c82f793d.

work. The scraper was created in python and scraped two websites. The first would collect all transaction hashes for a given contract. The second would provide the data for each transaction. Unfortunately, etherscan has Cloudflare protection on the site, meaning there is a cap to the number of requests the scraper can make before it is flagged and blocked. We found that we could scrape a transaction at 1.01 seconds and not be caught Cloudflare. An example data row can be found in 1.

Due to the number of transactions for some smart contracts (over 1000000), it would be unreasonable to expand the entire VeriSmart dataset to our format. Thus, this experiment would have to be performed on a few smart contracts at first and expanded over time if the results look good. There is an API that etherscan provides to download transaction data at 5000 transactions at a time, which could speed this process up by over 1000x; however, when I used the API it would sometimes provide no data and is too unreliable.

5 Conclusion

Our work plans to create a learned oracle for smart contract fuzzing. This would provide benefits over existing technologies and research as the fuzzing system would be scalable across more vulnerabilities, requiring less human effort to define oracles for all possible cases. The methodology presented leveraged publicly available data on the blockchain to form a dataset and train observer GAN models to understand the difference between user transactions (assumed to not exploit vulnerabilities) and randomly generated transactions that potentially exploit vulnerabilities. While the initial results of this work show promise, more sophisticated learning methods and potentially models will be required to remove the dependence of the model on having a large number of negative samples in the unlabeled data. Once a model is able to pass the initial proof-of-concept experiment with high AP even with a low percentage of negative unlabeled data, the full experiment can be implemented to understand how StateFuzz compares to existing fuzzing systems.

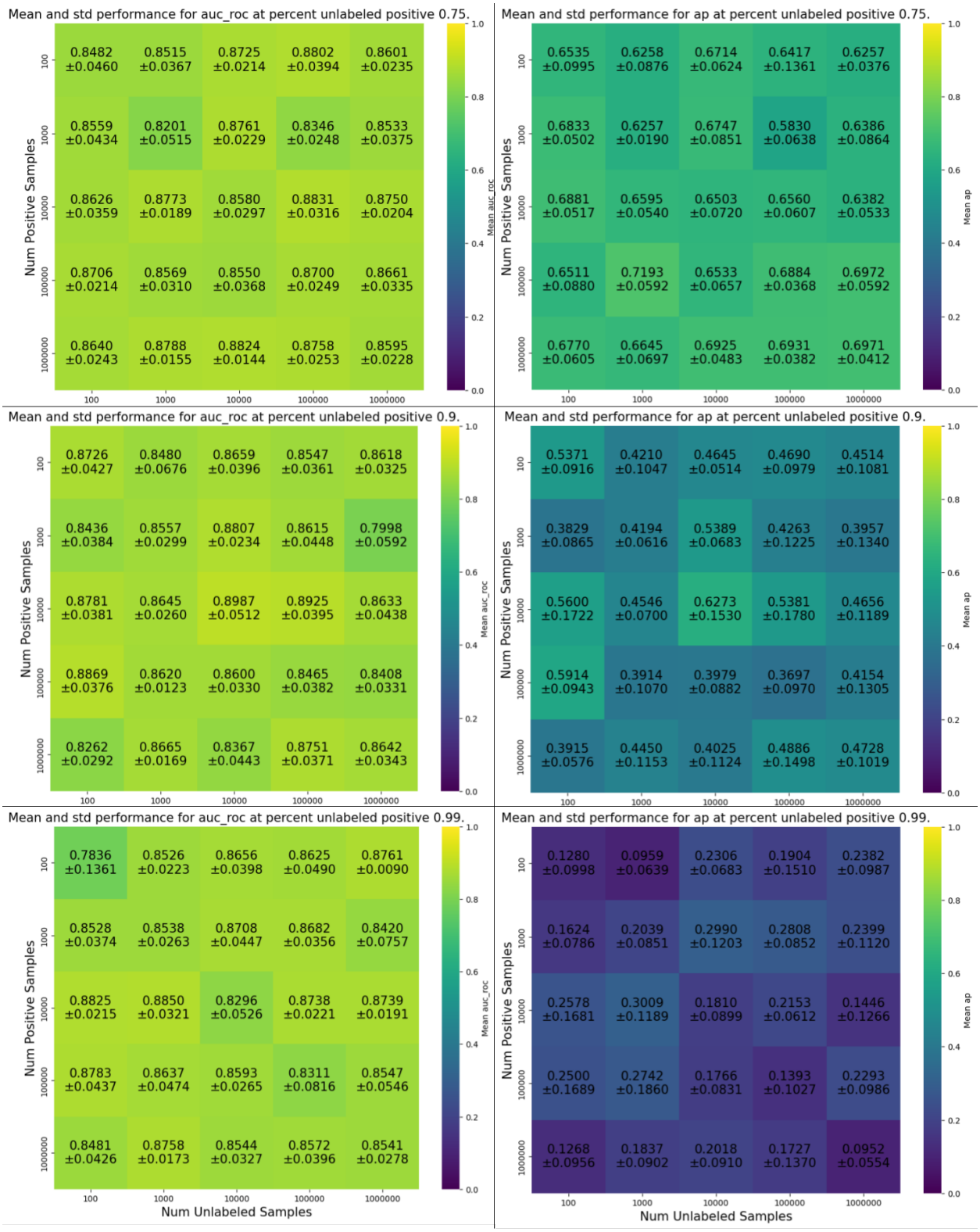

Figure 1: Metrics with varying number of positive samples, number of unlabeled samples, and percent of unlabeled samples that were positive.

References

[1] Analysis of the dao exploit, 2016.

[2] The parity wallet hack explained, 2017.

[3] What is fuzzing: The poet, the courier, and the oracle, 2017.

[4] Smart contract security in 2023: A simple checklist, 2022.

[5] Web3-native asset management is coming: Are institutions ready?, 2022.

[6] Smart contract security, 2023.

[7] Grieco, G., Song, W., Cygan, A., Feist, J., and Groce, A. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2020), pp. 557–560.

[8] He, J., Balunović, M., Ambroladze, N., Tsankov, P., and Vechev, M. Learning to fuzz from symbolic execution with application to smart contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (2019), pp. 531–548.

[9] Jiang, B., Liu, Y., and Chan, W. K. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (2018), pp. 259–269.

[10] Miller, B. P., Fredriksen, L., and So, B. An empirical study of the reliability of unix utilities. *Communications of the ACM 33*, 12 (1990), 32–44.

[11] Nguyen, T. D., Pham, L. H., Sun, J., Lin, Y., and Minh, Q. T. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (2020), pp. 778–788.

[12] Pei, S., Da Xu, R. Y., Xiang, S., and Meng, G. Alleviating mode collapse in gan via diversity penalty module. *arXiv preprint arXiv:2108.02353* (2021).

[13] So, S., Lee, M., Park, J., Lee, H., and Oh, H. Verismart: A highly precise safety verifier for ethereum smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)* (2020), IEEE, pp. 1678–1694.

[14] Weng, L. From gan to wgan. *arXiv preprint arXiv:1904.08994* (2019).

[15] Wüstholz, V., and Christakis, M. Harvey: A greybox fuzzer for smart contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2020), pp. 1398–1409.

[16] Zamzam, O., Akrami, H., and Leahy, R. Learning from positive and unlabeled data using observer-gan, 2022.

[17] Zou, W., Lo, D., Kochhar, P. S., Le, X.-B. D., Xia, X., Feng, Y., Chen, Z., and Xu, B. Smart contract development: Challenges and opportunities. *IEEE Transactions on Software Engineering 47*, 10 (2019), 2084–2106.