Uncovering Smart Contract VM Bugs Via Differential Fuzzing

Dominik Maier
dmaier@sect.tu-berlin.de
Technische Universität Berlin
Berlin, Germany

Fabian Fäßler
fabi@fabif.de
Technische Universität Berlin
Berlin, Germany

Jean-Pierre Seifert
jean-pierre.seifert@tu-berlin.de
Technische Universität Berlin
Berlin, Germany

ABSTRACT

The ongoing public interest in blockchains and smart contracts has brought a rise to a magnitude of different blockchain implementations. The rate at which new concepts are envisioned and implemented makes it hard to vet their impact on security. Especially smart contract platforms, executing untrusted code, are very complex by design. Still, people put their trust and money into chains that may lack proper testing. A behavior deviation for edge cases of single op-codes is a critical bug class in this brave new world. It can be abused for Denial of Service against the blockchain, chain splits, double-spending, or direct attacks on applications operating on the blockchain. In this paper, we propose an automated methodology to uncover such differences. Through coverage-guided and state-guided fuzzing, we explore smart contract virtual machine behavior against multiple VMs in parallel. We develop *NeoDiff*, the first framework for feedback-guided differential fuzzing of smart contract VMs. We discuss real, monetary consequences our tool prevents. NeoDiff can be ported to new smart contract platforms with ease. Apart from fuzzing Ethereum VMs, NeoDiff found a range of critical differentials in VMs for the Neo blockchain. Moreover, through a higher-layer semantics mutator, we uncovered semantic discrepancies between Neo smart contracts written in Python when executed on the blockchain vs. classic CPython. Along the way, NeoDiff uncovered memory corruptions in the C# Neo VM.

CCS CONCEPTS

• **Security and privacy** → *Domain-specific security and privacy architectures*; **Software and application security**.

KEYWORDS

Differential Fuzzing, State-Aware, Smart Contract VM

ACM Reference Format:
Dominik Maier, Fabian Fäßler, and Jean-Pierre Seifert. 2021. Uncovering Smart Contract VM Bugs Via Differential Fuzzing. In *Reversing and Offensive-oriented Trends Symposium (ROOTS'21), November 18–19, 2021, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3503921.3503923

ROOTS'21, November 18–19, 2021, Vienna, Austria

ACM ISBN 978-1-4503-9602-8/21/11…$15.00
https://doi.org/10.1145/3503921.3503923

1 INTRODUCTION

Blockchains and cryptocurrencies are often associated with money and investments. This paper focuses on a very different aspect to them, the underlying **smart contract virtual machines** and their security. Systems like Ethereum [3] and Neo [10] extend the concept of a traditional blockchain with such a built-in virtual machine: Turing-complete, yet limited by artificial mechanics, to guarantee the program's termination. Every execution of the contracts is publicly recorded, and every node can run each smart contract to verify the transactions. For computer security research, they introduce a new and foreign execution environment when compared to traditional CPUs and operating systems. Like in an ordinary OS, the virtual machine executing the smart contract defines an interface to interact with various components. Without files, sockets, and threading, but with direct access to the blockchain, smart contract VMs create weird computing environments to explore and research. From these new platforms, new security issues and bug classes arise.

For common chains, all participating smart contract VMs need to behave consistently. As we show in this paper, even the slightest difference in op-code behavior can have drastic consequences. They reach from stolen money, over chain splits, to the complete collapse of the ecosystem. To mitigate security bugs before they reach production, we create NeoDiff, a purpose-built tool to uncover differences automatically. It leverages differential fuzzing, generating valid input and feeding it back to the target to find differences in an automated fashion. In over 2.5k LOC, NeoDiff packs a fully-featured tool for security research. On top of coverage, a common feedback mechanism for fuzzing, NeoDiff can take state progression into account, backed by the virtual machine. While diffing the two largest virtual machines for Ethereum only uncovers false positives, likely due to rigorous manual testing, we uncover a wide variety of bugs for the other ecosystem we test: we take a deep dive into virtual machines for Neo, a less researched chain, albeit still spanning a 24-hour trading volume of 1.5 bn dollars in early 2021. Here, NeoDiff finds critical bugs, including differences between alternative VMs, as well as memory corruptions in the pre-release code of Neo's main chain. Since the Neo ecosystem offers the option to write smart contracts in traditional programming languages, such as Python, they open the door for further fuzzable differences: python's language semantics.

In contrast to traditional fuzzing, the condition a differential fuzzer strives to trigger is not corrupted memory or a crash [14], but a difference in output or state. NeoDiff can compare targets of different ecosystems and programming languages, such as NeoPython and the Neo VM in C#, and support feedback mechanisms that go beyond code coverage. NeoDiff found a wide range of differences

between the Neo VM implemented in C#—used by the main consensus nodes—and the *neo-python* VM implemented in Python—used by many client programs that want to interact with the blockchain, such as wallets or web applications [21]. Moreover, NeoDiff uncovered semantic differences between CPython and *neo-boa* Python smart contracts for Neo, as well differences on opcode-level and memory corruptions in its VM, written in C#.

Our contributions are as follows:

- We develop, evaluate, and open-source *NeoDiff*, a purpose-built platform for differential fuzzing of smart contract VMs. NeoDiff is a generalized framework for differential, feedback-driven fuzzing, applicable to a broad range of VMs.
- We implement NeoDiff back ends to differential fuzz the *openethereum* against the *geth* Ethereum VM, as well as the *Neo VM* against *neo-python*.
- As an additional target, we diff the CPython language semantics against smart contracts written in Python, finding semantic gaps in smart contract programming languages and the associated security ramifications.
- We discuss how discrepancies in smart contract VMs can be exploited beyond the blockchain network itself. For example, we describe how to attack applications built on top of the blockchain.
- NeoDiff helped find and fix critical bugs in the Neo smart contract ecosystem, including differences and: memory corruptions.

2 BACKGROUND

We are convinced building tooling applicable spanning multiple smart contract platforms will add valuable insight for the research community. In this section, we take a closer look at the Neo smart contract platform. By the time of writing this paper, the Neo blockchain had a reported market cap of about 3.7 billion dollars and a 24h trading volume of over 1.5 billion dollars.

2.1 The Neo Smart Contract Ecosystem

Many people trust the Neo ecosystem to handle monetary assets. Neo started to get research attention, for example, through work by Qin et al., analyzing Neo's dBFT scheme [24]. Neo is a proof-of-stake blockchain using a small number of consensus nodes by design, voted on by its stakeholders. As opposed to Ethereum, which uses purpose-built languages like Solidity for smart contracts, Neo offers front ends for a variety of traditional languages. The Neo homepage lists support for 8 major programming languages, including *Python*, *Go*, *JavaScript*, *Java* and *C#*, with an additional planned support of *C* and *C++*. The Neo compilers are typically not full compilers but use the languages' official compilers to transform code into an intermediate representation, which is then translated to *AVM* code, the bytecode for the Neo VM. The main consensus nodes run the official VM, written in C#, and make up the core Neo blockchain network. To interact with the blockchain—for example, to send transactions or to invoke smart contracts—users need to use client applications. The Neo ecosystem offers SDKs for developers to build such applications on top of the Neo blockchain and to interact with smart contracts. The SDKs have to understand all the transactions in the Neo blockchain and thus include their own Neo VM implementation to execute the smart contracts locally.

Neo Assets The native assets of the Neo blockchain are NEO, the main currency, and Neo GAS (GAS), primarily used to pay for the deployment and execution of smart contracts. NeoGAS is being generated and distributed over time to Neo wallets, proportionally to the amount of Neo owned. Along with the typical verification of asset transactions, each node also includes the Neo VM, able to execute smart contracts in the form of bytecode. A certain GAS cost is associated with each opcode and syscall. The consensus algorithm used by the consensus nodes is called *delegated Byzantine Fault Tolerance* and is based on *Practical Byzantine Fault Tolerance* [7, 16]. The following sections will briefly introduce the Neo smart contract ecosystem.

The Neo Virtual Machine The Neo VM is responsible for executing smart contracts uploaded to the Neo blockchain. Every participant in the blockchain network, including wallets and exchanges, needs their own Neo VM to execute contracts. In contrast to traditional process VMs, Neo VMs don't provide access to hardware or the filesystem but interact with data on the blockchain. They abide by the Neo bytecode format and provide APIs to access, for example, permanent storage and blockchain transaction data. The official Neo VM used by the consensus nodes is implemented in C# [9]. Client programs that want to interact with the blockchain, such as wallets or web applications, often make use of alternative VMs, matching their respective programming languages. For example in go and Python.Evaluating smart contracts locally allows clients to read data from a contract's storage directly and to react to events without having to query slow APIs or trusting a centralized node.

The C# VM The architecture of the main VM is depicted in Fig. 1.

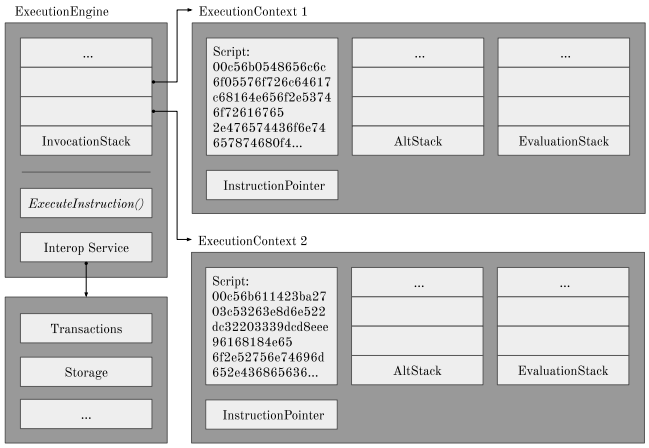


Figure 1: Neo VM architecture: The main component of the Neo VM is the *ExecutionEngine*. It executes each smart contract in independent *ExecutionContexts*, ensuring strict separation between smart contracts. Each contract can work on two seperate stacks, the *AltStack* and *ExecutionStack*. Through the *Interop Service*, contracts can access external data and state.

The *ExecutionEngine* is the core of the Neo VM and connects the

various components. It also contains the *ExecuteInstruction()* function that will interpret and execute any opcode. When a contract is executed, an *ExecutionContext* is created and stored in the *InvocationStack* of the ExecutionEngine. This ExecutionContext contains the *Script*, the bytecode of the smart contract, as well as two stacks. The two stacks are the *EvaluationStack*, for storing results, and *AltStack* for temporary data. For each call to other contracts, a new ExecutionContext is created and placed on the InvocationStack. This means the contracts are not sharing their EvaluationStack or AltStack with each other, which is a beneficial security property.

Persistent Storage Most data, such as blockchain transactions and the contract's permanent key-value storage, is accessed through the *SYSCALL* opcode. The permanent key-value store, or rather the values themselves, are not directly stored in the blockchain. The blockchain only records transactions, which include transactions that call smart contracts. Executed smart contracts can read, write and delete data in this storage. In order to get the current state of the storage, all transactions have to be replayed, and smart contract invocations need to be executed. The storage might store authentication information or balances of tokens. Giving another contract access to this storage context object authorizes this other contract to access the same storage. This creates an attractive attack surface for Neo smart contracts.

Storage-related issues could stem from bugs in the implementation itself, or programmers might carelessly authorize untrustworthy contracts. The potential impact of developers willingly authorizing another contract is similar to what can happen with *DELEGATECALL* in Ethereum smart contracts [13]. This call will also execute another contract and authorize (delegate) the other contract to interact with the same storage.

3 RELATED WORK

Differential fuzzing has been used to test compilers, and libraries since before the concept of smart contracts were envisioned.

CSmith A notable example for differential testing is CSmith by Yang et al., which auto-generates C programs and hunts for differences in program behavior compiled with different compilers and optimization levels. It found a wide variety of bugs in different C compilers [25].

DifFuzz With DifFuzz, Nilizadeh et al. propose the use of differential fuzzing to find side channels by observing the program behavior concerning a resource [17].

NEZHA Petsios et al. proposed NEZHA, a generalized approach to differential fuzzing of conventional binaries, which is more efficient than existing tools at finding bugs in complex software, targeted mainly towards binary libraries. Instead of using the code coverage of a single target as feedback, they use multiple feedback maps for the targets under test [23].

HyDiff HyDiff by Noller et al. makes use of similar concepts as NEZHA, for example, tracking changes of branch coverage in one of the targets under test, but also makes use of symbolic execution and static analysis. The tool has a range of divergence heuristics to improve the selection of the mutated inputs [18].

EVMFuzz *EVMFuzz* [12] is a fuzzer that was proposed by Fu et al. for differential fuzzing smart contract VMs. It generates contracts and feeds them into multiple Ethereum VMs, in order to find their discrepancies. The tool works by mutating smart contracts on the Solidity language level.

4 NEODIFF

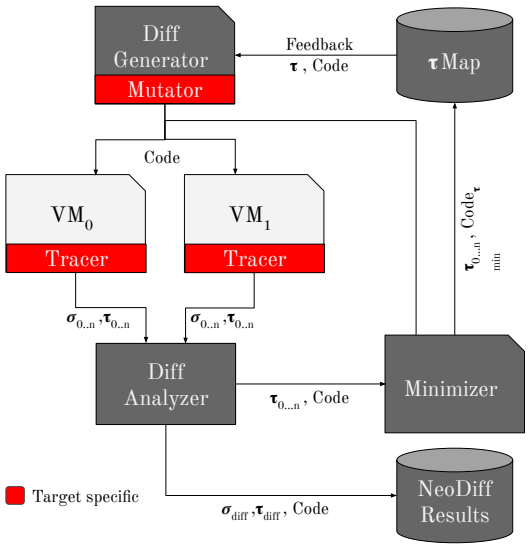


Figure 2: NeoDiff Building Blocks and Data Flow

This section discusses the concepts behind *NeoDiff*. The *NeoDiff* fuzzer performs differential testing [15] on alternative implementations of the same VM in order to find discrepancies between their execution.

4.1 Design

NeoDiff is feedback-guided yet still able to explore multiple tested virtual machines at the same time. With options for different lightweight and portable feedback mechanisms, we can adapt NeoDiff to VMs written in a range of different languages across multiple smart contract platforms. NeoDiff is purpose-built from scratch to a) be quickly adaptable to multiple targets b) mutate complex bytecode c) work with additional state feedback and waypoints.

In contrast to prior work, such as EVMFuzz, NeoDiff takes a lower-layer approach, generating valid opcode sequences through feedback-based mechanisms directly. Consequently, our approach will emit any possible opcode sequence, while the Solidity compiler may not emit sequences that don't have a meaning on its higher language level. We expect smart contract VMs to be less tested against non-standard opcode sequences that are never emitted by higher-level languages.

In the following sections, the important pieces of NeoDiff, as depicted in Fig. 2 will be discussed in detail.

Diff Generator The *Diff Generator* generates code to be executed by the VMs. For raw bytecode, the feedback-based mutator will yield good results, applying a set of useful mutations to the test cases, such as splicing and appending of high-coverage input, random byte flips, and more. In comparison to language-level fuzzers, like EVMFuzz [12], bytecode level mutators are also able to

find uncommon and invalid bytecode sequences that the high-level language compiler will never emit.

Mutators It's possible to specify custom mutators to customize the fuzzer for specific targets. By default, NeoDiff will apply one of the following mutations in a loop, starting from an empty input, until it reaches a minimum contract length:

- *Random Bytes*: A random amount (0x1 - 0xF) random bytes get appended to the contract.
- *Full Splice*: Parts of a random second contract are appended. Favors test cases that produced new coverage.
- *Partial Splice*: Parts of a random second contract are appended. Favors test cases that produced new coverage.
- *Byte Insert*: NeoDiff inserts a random byte at a random place into the current contract.
- *Special Operations*: Special important VM opcodes, such as `PUSHBYTES` for ETH, are pushed.

In contrast to existing coverage-guided fuzzers, like AFL++*aflpp*, neo-diff can randomly generate completely new contracts, with the goal of not getting stuck on a local maximum.

The NeoDiff Evaluator will then extract bytecode sequences of successful runs to be used by the next NeoDiff Generator run.

On top of the predefined mutations, it's easy to define chain-specific mutators. As a concrete example, for semantic differences in Python, we implement a custom mutator that generates Python code. It emits valid Python code, each block according to a state value, which may be mutated. The generated code from the Diff Generator is then passed on to both VMs to be executed.

Diff Analyzer The instrumentation of the VMs must be adapted to produce a valid trace for the *Diff Analyzer*. The trace returned from the run of a VM is a list of executed opcodes together with a so-called type-hash τ and a state-hash σ. The Diff Analyzer can then compare σ of both traces to find diffs and use τ to classify if new states were reached. We assume a state to diverge if any state-hash σ during execution differs between VMs, for the same input. **Minimizer** In case of a diff, the offending opcode included in τ, as well as the code that caused the diff, is stored in *NeoDiff Results*. The part of the code that executed cleanly is forwarded to the *Minimizer*, together with any τ of this run. The Minimizer uses the instrumented VMs to find the shortest byte sequence that results in a specific τ and stores it in the τ Map. The τ Map is growing over the course of the fuzzing session and can be used by the Diff Generator as feedback for further mutations.

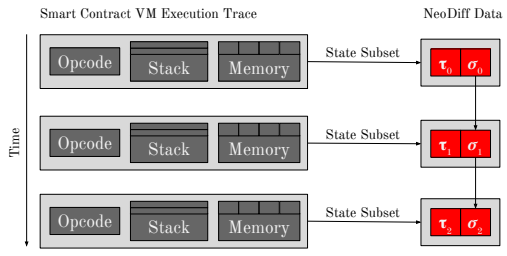


Figure 3: Relation between VM internal execution states and the NeoDiff type-hash τ and state-hash σ

4.2 State-Hash

To identify differential execution, NeoDif makes use of a so-called state-hash σ. Fig. 3 shows that σ will take a subset from the current state during execution, as well as every previous state. A researcher has to define which information is important for a diff and include it in the hash. In a stack-based VM, this includes a probabilistic sample of the stack; if there are registers, σ should include register values, and if there is additional memory, this memory should be taken into consideration. Of course, there is a trade-off between execution speed and precision that has to be taken into account. The state-hash σ must be continuously updated, for each opcode, if possible. NeoDiff then uses the state-hash to detect diffs in any of the executed operations.

4.3 Type-Hash

As can be seen in Fig. 3, type-hash τ also uses state information from the execution trace, but it is not chained with previous states during execution—it only represents a single opcode state. The name *type-hash* stems from their purpose in the Neo VM, as the Neo VM supports various types such as Integer, ByteArray, Array, Boolean, Map, and Struct. Even though EVM only supports 256-bit ints as types, we derived *virtual* types, such as valid addresses, small ints, bools, that are known to lead to other executions inside opcodes. While NeoDiff can also work on traditional code coverage, if available, the type-hash τ helps to sort and categorize diffs and allows to quickly recognize patterns—for example if all diffs happened due to the top stack item being of a certain type. For the type-hash, NeoDiff uses the type of the top two stack items, prefixed with the current opcode. In Neo, if the *ADD* instruction is executed while an Integer (1) and a ByteArray (2) are on the stack, the type-hash τ of this moment would be *ADD_12*.

Type-hashes are similar to previously proposed human-guided feedback mechanisms like waypoints by Padhye et al. [22] and annotations for Ijon, by Aschermann et al. [2], although both increase the engineering overhead, while τ is specifically designed as a lightweight coverage metric for differential VM fuzzing and other exotic targets. Instrumenting VMs, written in a range of programming languages, with full code coverage information per opcode can be an additional engineering task, whereas type-hashes work well across virtual machines for the same smart contract platform, making them a good fit for this use case.

To evaluate the effectiveness of type-hashes, we tested how many times a newly-found type-hash results in improved code coverage of the underlying VM. For this reason, we instrumented the neo-python VM with code tracing, logging the execution trace for each opcode and their type-hash. The Fig. 4 shows for each opcode the ratio between finding a new type-hash and it also being new code coverage. It shows that finding a new τ in 86% of cases (median) also reaches new code coverage internally, making it a beneficial feedback channel.

Still, this does not imply that type-hashing will pick up all changes in coverage for each of the tested VMs. Using actual code coverage as τ is supported by the diff engine, and test cases can be shared with AFL++ [6].

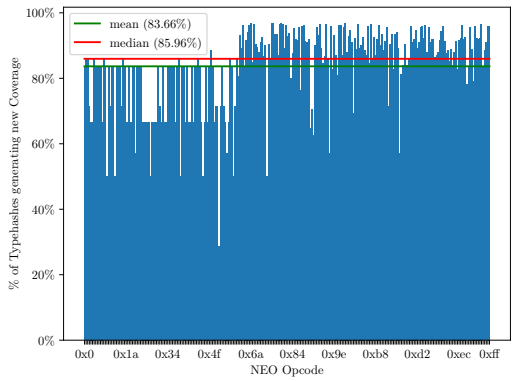


Figure 4: Percentage of new type-hashes reaching new coverage per opcode in neo-python: over 83% of new type-hashes also lead to new coverage.

4.4 Minimization

After bytecode executions, NeoDiff collects the feedback. When it discovers a new type-hash τ, it minimizes the code to reach this τ. The goal is to use the shortest possible smart contract with the same semantics, as the contracts tend to grow fairly big otherwise.

The minimization re-runs the VMs with an increasing number of bytes of the test case. For each byte, it checks if any of the new τ are included in the trace and stores them in the τ Map. Once all type-hashes τ are found during execution, we assume that the test case is minimal. This byte-for-byte minimization makes the minimization part of NeoDiff portable to other VMs. An issue is that bytecode might not execute linearly due to jump opcodes. This could result in the opcode of interest that leads to a new type-hash, to sit in the middle of the code. Though most opcodes are executed linearly and thus the best-effort approach should suffice.

4.5 Diff Triaging

Fuzzing VMs for differentials is an iterative process.To take the burden off of triaging, Neo Diffs groups diffs based on the type-hash τ of a potentially offending opcode. Simply re-running the VMs and looking at the produced states allows very easy verification of these diffs. For example, through NeoDiff we quickly found *go-ethereum* and *openethereum* return different values for opcodes like *GASLIMIT*. However, it was a false-positive in this case, caused by different gas limit default configurations in each chain. The security researcher using NeoDiff can then either ignore false positives or adjust the fuzzer configuration.

4.5.1 C# Neo VM against neo-python. To target the Neo VM, we included the current opcode and stack values in each step of the execution in the state-hash σ. To instrument the Neo VM with a tracer that produces output for the Diff Analyzer, we call each VM's *ExecutionEngine* directly. As both the Python VM and the C# VM share a similar code design, this approach works for both. As Neo VM knows different types, τ was chosen to use the opcode and the type of the topmost stack elements.

4.5.2 Geth against Openethereum. To instrument the Ethereum VM, we feed the existing JSON trace outputs from go-ethereum(*geth*) and openethereum (formerly parity) into NeoDiff. In contrast to the Neo VM, the Ethereum VM does not have a concept of types. Everything is encoded in *uint256* values. So we instead created τ based on heuristical *virtal types*. Though in this case, it is still possible to define *virtual types* to serve a similar purpose. For example, if a given value on the stack looks like a potential address, we assign it the virtual type of *Address*. Similarly, we interpret values that are higher or lower than addresses as another virtual type, that potentially leads to other code paths in the opcode execution handlers. In total, for EVM, we deduct the following 6 *virtual types*, based on value ranges:

(1) **Empty** Value not set
(2) **Boolean** Values 0 or 1
(3) **Ethereum Address** Exactly 160 bit
(4) **ByteArray** More than 160 bits
(5) **Large Number** More than 64 bits
(6) **Number** All other values

Using these values as τ, we were able to plug Ethereum into the differential fuzzer NeoDiff, using only a few lines of additional tracing code and some custom mutations.

4.5.3 CPython aginst Neo Python. We developed a custom mutator for NeoDiff, which is able to produce valid Python scripts that can also run as valid Neo smart contracts with various random expressions, followed by a return of a unique random identifier and whatever value was calculated before. For Python diffing, the state-hash consists of the hash of artificially crafted return tuples, containing the value calculated in the previous expression and a unique id of the executed code path.

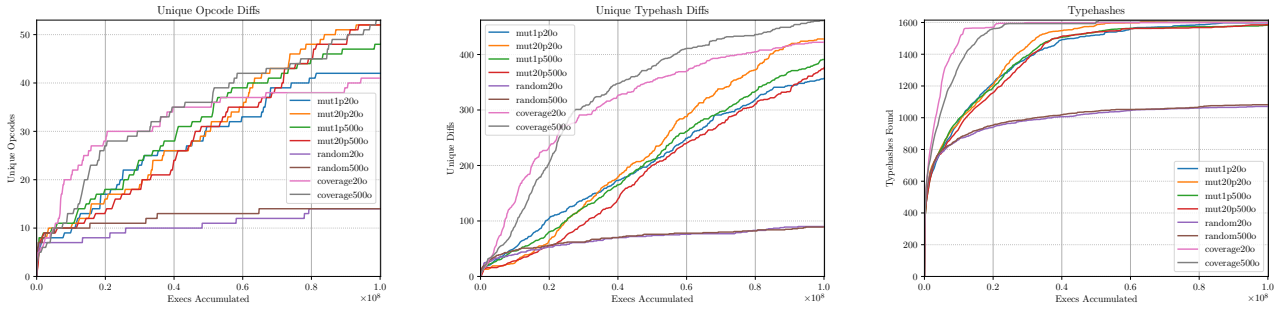
5 EVALUATION

To test the differential fuzzer NeoDiff, we implemented target-specific code for multiple targets. In total, we ran NeoDiff against 3 Targets, with 2 VMs each: the *Neo C# VM* against *neo-python*, and the two largest Ethereum VMs, *geth* (go) and *openethereum* (rust). As the third backend, we pivoted NeoDiff to fuzz language semantics. Namely, we diff CPython against Neo smart contracts written in Python and compiled with *neo-boa* to the Neo VM.

5.1 C# Neo VM against neo-python

We fuzzed *neo-python:v0.9.1* against the *neo-vm:2.4.3* C# VM. To instrument the Neo VM with a tracer that produces output for the Diff Analyzer, NeoDiff executes each VM's *ExecutionEngine* directly. We patched in a persistent mode, receiving inputs from NeoDiff without the need to respawn the VM, allowing us to reach thousands of execs per second. The execution speed depends heavily on the length and kind of contracts currently emitted (i.e., loops drag down the execution speed). As Neo VM knows different types, we set τ to use the opcode and the type of the topmost stack elements.

Using Neo as a base, we evaluated four mutation strategies, each in a 20 opcode, *20o*, and a 500 opcode, *500o*, variant - stopping the execution of the VM at a maximum of 20, or 500 executed opcodes, respectively. All mutation strategies find diffs, with the purely random, *random20o* and *random500o* finding the most differences at first glance. However, digging deeper, the pure-random diffs are not



(a) Diffs on Opcodes **(b) Diffs with new τ** **(c) Total τ found in Neo**

Figure 5: Fuzzing Results for multiple NeoDiff runs of neo-python and the main Neo VM. We compare multiple mutators and purely random generation, as a baseline

diverse, as seen in Fig. 5a, after days of runtime, never exceeding 22 unique diffing opcodes.

The differentmutation strategies we evaluated are:

(1) **random** True-Random fuzzing with fully random bytes, used as a base reference
(2) **mut1p** Hand-written custom mutator for Neo, with snippets to push special values to the stack. With low probability, it will use the coverage τ-feedback.
(3) **mut20p** Same mutations as *mut1p*—but will use feedback-based mutations with 20 times the initial probability.
(4) **coverage** NeoDiff default mutation strategy, with initial random byte sequences, mostly relying on coverage τ-feedback with random byte flips and splicing.

True random also lags behind all other on type-hashes, as seen in Fig. 5c indicating the success of our mutation strategies. At this metric, the type-hash-guided variants do very well. Fig. 5b, showing the diffs with unique type-hash proves that the pure-random variants cannot compete against the custom mutators and coverage-guided mutators. We draw another conclusion regarding the size of the test case: while the *20p* variants are faster initially, they get overtaken by their larger variants eventually—probably because complex operations cannot be expressed in 20 opcodes easily. All mutation strategies found a wide range of real diffs in Neo. They are further discussed in-depth in Sect. 6.

5.2 Geth against Openethereum

For EVM, we perform the evaluation using the best mutation strategy according to our prior experiments on Neo: a single mutator with a very high likelihood of feedback-based mutations. State-hashes σ for EVM include the opcode, the stack, the memory, and the gas cost. A difference in any of these would result in an exploitable discrepancy. To instrument the Ethereum VM, existing JSON trace outputs from go-ethereum(*geth*) and openethereum (formerly parity) can be reused and fed into NeoDiff.

All in all, the Fuzzer found six instances on different opcodes, see Fig. 6a. After triaging, all differences could be traced back to different behavior on empty chains between *geth* and *openethereum*. We could not identify any security-critical differences for existing chains. More details about the diffs can be found in the discussion in Sect. 6.3

5.3 CPython aginst Neo Python

To diff Python code, we developed a grammar-based custom mutator for NeoDiff, not based on τ, which is able to produce valid Python scripts that can also be run as valid Neo smart contracts. We could not simply get an opcode trace, as the executed CPython bytecode is fundamentally different from Neo bytecode. Instead, for Python diffing, the state-hash consists of the hash of artificially crafted return tuples, containing the value calculated in the previous expression and a unique id of the executed code path.

NeoDiff was able to find several semantic differences for the Python language. After over 20k executions, NeoDiff had already identified a range of semantic diffs between CPython and Neo Smart contracts:

neo-python: **14 diffs** vs. Python2.7 | **13 diffs** vs. Python3.7
Neo VM: **11 diffs** vs. Python2.7 | **10 diffs** vs. Python3.7

In fact, what can be seen here is an unintended additional discovery of regular VM diffs between C# and neo-python. We explain the security impact of semantic differences for smart contracts in Sect. 6.4.

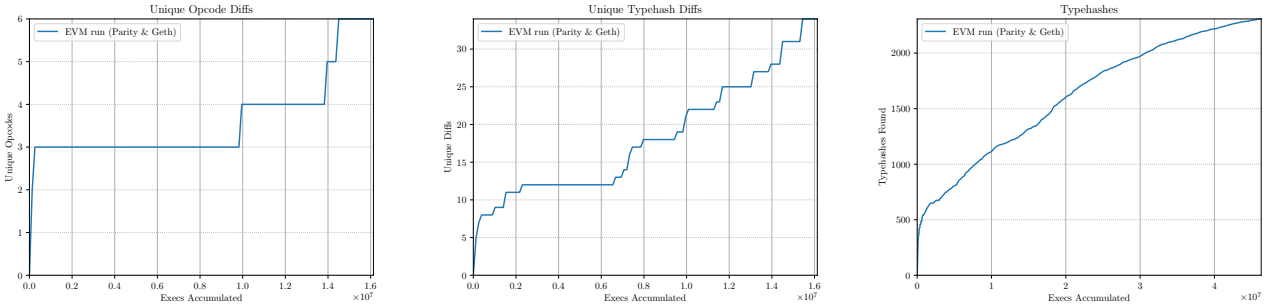
6 DISCUSSION

In the following section, we present an in-depth security analysis of the fuzz targets after analyzing the results of NeoDiff.

6.1 Security Impact of Differences in Smart Contract VMs

To understand why differences between smart contract VMs can be classified as security-critical bugs, this section will briefly introduce three categories of differences, together with their possible attacks.

6.1.1 On-Chain Diff. On-Chain diffs describe diffs between VMs that are deployed as the participating nodes of a blockchain network. In the case of Ethereum, this refers to the miners who have to confirm all transactions. For each execution of a smart contract on the chain, the bytecode runs on all nodes that need to agree



(a) Diffs on Opcodes **(b) Diffs with new θ** **(c) Total τ found in EVM**

Figure 6: The results of our fuzzing run, comparing the Parity and Geth EVM VMs

with the propagated result in the chain when a block is forged. Implementations between nodes on the chain must never diverge. In a proof-of-work-based chain like Ethereum, a diff between alternative implementations with many active nodes can lead to a split of the overall hashing power. Depending on the bug, the VM implementation used by the majority mining power will keep the main chain going, while the miners using the minority implementation may not agree and will therefore split off the chain, with a different result for this transaction.

6.1.2 Blockchain Application Diff. The issue described in the previous section can further be applied to applications built upon the blockchain—including wallets, exchanges, DApps, and more. These applications must understand new transactions and verify their correctness by executing them. Local execution results must match those on the main network. Else the application will suffer from a local chain split, resulting in a "wrong" view of the blockchain's state.

6.1.3 Contract Semantics Diff. Language semantic differences are a discrepancy between the smart contract programming language implementation and the original programming language. Semantic diffs happen if smart contracts can be written in languages borrowed from other execution environments. In the case of Neo, smart contracts can be implemented in many different languages, for example, Python. If the behavior differs from the spec, developers may introduce bugs. Their expectation of how the code should behave differs from the actual implementation. A malicious developer can abuse these different behaviors to hide backdoors in plain sight, as even experienced Python developers have no chance to spot them.

6.2 PoC Attack on Neo VM Differences

To show why VM discrepancies in Neo are an actual security issue, we present an example attack for blockchain application diffs.

First, we set up *neo-local* [19], a project to run a personal Neo blockchain in Docker containers. It sets up a private Neo network for testing, using the official C# node [8]. It also exposes a *neo-python* [21] prompt to be used as a client to interact with the blockchain. We set up a project that will use the Python VM to execute the contracts locally and a C# VM that will be executing contracts on the nodes. The attacks can be reproduced using the version *neo-local:0.12*. With the *neo-python* prompt, it is possible to compile and deploy the contract to the private chain. We depict this in Listing 1. The resulting *SmartContract.Contract.Create* transaction event also shows the deployed contract address, as well as the transaction identifier (line 8). After we deploy the contract, we can invoke it using its address. A local test invocation, performed by neo-python using the Python VM, returns the hex-encoded string *Python*. After providing the wallet's password, the transaction is sent to the actual private blockchain network. Once the transaction is broadcasted, the local wallet using *neo-python* will receive the new transaction and execute it locally. As can be seen, the result of the contract is indeed the string *Python* (line 22).

```
1  neo> wallet open neo-privnet.wallet
2  neo> sc build /smart-contracts/div.py
3  Saved output to /smart-contracts/div.avm
4  neo> sc deploy /smart-contracts/div.avm False False False 00 07
5  Please fill out the following contract details:
6  [Contract Name] > DIV PoC
7  [...]
8  [I 191006 13:24:46 EventHub:62] [SmartContract.Contract.Create][6562] [0
       f904e38fb40891daf11685a29f25a6716d331e7] [tx 0
       c86ab2fcce081f28259a9018e78dc8f8681046dfa06f414df4e521e28978657] {'
       type': 'InteropInterface', 'value': {'version': 0, 'hash': '0
       x0f904e38fb40891daf11685a29f25a6716d331e7', 'script': '58c56b516a0052
       [...]6c7566006c7566', 'parameters': ['Signature'], 'returntype': '
       String', 'name': 'DIV PoC', 'code_version': '', 'author': '', 'email':
        '', 'description': '', 'properties': {'storage': False, '
       dynamic_invoke': False, 'payable': False}}}
9  [...]
10 neo> sc invoke 0x0f904e38fb40891daf11685a29f25a6716d331e7
11 [...]
12 -------------------------------------------------------------
13 Test invoke successful
14 Total operations: 39
15 Results [{'type': 'ByteArray', 'value': '507974686f6e'}]
16 Invoke TX GAS cost: 0.0
17 Invoke TX fee: 0.0001
18 -------------------------------------------------------------
19 [password]> ***
20 [...]
21 Relayed Tx: 7740d42b12ecf4f7c9d92222b6eb629a2757c37e47b2cb6dffca3242b7ed98d8
22 [I 191006 13:25:03 EventHub:62] [SmartContract.Execution.Success][6565] [0
       f904e38fb40891daf11685a29f25a6716d331e7] [tx 7740
       d42b12ecf4f7c9d92222b6eb629a2757c37e47b2cb6dffca3242b7ed98d8] {'type':
        'Array', 'value': [{'type': 'ByteArray', 'value': b'Python'}]}
23 [...]
24
```

Listing 1: DIV test contract deploy

To see what happened in the actual node running the C# Neo VM, the JSON RPC endpoint exposed by one of the main nodes can be used. This will return the result hex-encoded *"437368617270"*—it returned the string *Csharp* instead.

Locally, the Python VM came to a different result when executing the contract, compared to the real C# VMs running on the consensus nodes!

Simple Diff Listing 2 shows a Neo smart contract exploiting a difference in integer division identified in Sect. 6.5.

```
def main():
    a = 1
    b = -2
    c = a/b
    if c == -1:
       return "Python"
    else: # == 0
        return "Csharp"
```

Listing 2: DIV discrepancy exploit in a Python smart contract

The code in this minimal example will return the string *"Python"* or *"Csharp"* depending on the VM execution context. It exploits an integer division difference. This would already suffice to confuse neo DApps. We list other differences in Neo, found by NeoDiff, in Sect. 6.5.

Complex PoC Attack For our PoC, we target a typical application that uses *neo-python* to react to certain events of the blockchain, such as crowdsales, general operations of tokens, or even exchanges. We successfully implemented this attack.

There are three components to this attack: the attacker's smart contract, the attacked *PWN token* representing any token smart contract, and *server.py*, an application built using *neo-python* for smart contract evaluation.

The *server.py* interacts with PWN token. For that, it uses *neo-python* to subscribe to events created by the PWN token contract. The script listens for *transfer* events, specifically those events where tokens are sent to the owner of this contract. The idea of this application is to facilitate refunds of bought tokens—essentially selling PWN tokens back to the owner in exchange for NEO. Because Neo smart contracts can receive NEO but not send it with smart contract code, transferring NEO has to be done either manually or by using an external application, like *server.py* in this example.

We provide the relevant code in Appendix A.

(1) **attack.py** The attacker smart contract
(2) **nep5.py** PWN token representing any token smart contract
(3) **server.py** - An application built with *neo-python* as the target

Listing 3 shows how to setup and run the correct version.

```
root@host $ git clone https://github.com/CityOfZion/neo-local/
root@host $ cd neo-local
root@host $ git checkout 0.12
root@host $ make pull-images
root@host $ make setup-network

```

Listing 3: Setup *neo-local* for PoC attack

The PWN token in our PoC attack is based on an example token smart contract for Neo and follows the NEP-5 token standard [1]. We extended the contract with usual functionality: to *mint* initial tokens for the owner and to be able to *buy* tokens from the owner with NEO.

After the contract is deployed, the *server.py* can be launched. The owner in this example is the address `AK2nJJpJr6o664CWJKi1QRXjqeic2zRp8y`. We assume a user bought PWN tokens with NEO previously but would like to return them now. Once the user transfers PWN tokens back to the owner, the *server.py* script gets notified and executes a transaction to send an equivalent amount of NEO back to the sender.

Alice will serve as the attacker. She will buy 1bil PWN tokens by invoking the *buy* operation and attaching 10 NEO to the transaction.

Summarized, the steps so far are as follows:

- PWN token contract is deployed and initialized
- server.py is observing PWN token transfer events
- Alice bought 1bil PWN tokens for 10 NEO

Alice can now implement a malicious attacker contract using the discrepancy in the integer division. The *attack.py* contract in Listing 4 will initiate a token transfer from a given address to a different address. Depending on the execution context, it will use the address given in *real_to* or *fake_to*. **Attack Execution** After Alice has deployed the attack.py contract, Alice will transfer her 0.1bil of PWN tokens to this contract. This transfer is necessary because the *attack.py* contract will initiate the token refund and thus needs to own tokens to transfer them. When Alice now invokes the attack contract, different logic is executed depending on the VM environment. In neo-python, used by *server.py*, it will create a token transfer back to the owner—thus triggering a refund. In the actual chain using the C# Neo VM, however, the tokens are simply transferred back to Alice!

Once the attack is complete, the balance checks using *neo-python* shows that Alice has only 0.9bil PWN tokens left and the 0.1bil were successfully returned to the owner. This means that server.py will refunded 1 NEO back to the attacker in the actual chain, even though no tokens arrived on the main Neo chain, due to the VM difference, found by NeoDiff. Indeed, querying the C# VMs' JSON RPC endpoint reveals that Alice still owns all her initial 1.0bil PWN tokens. However, the NEO transfer still occurred because the refund application assumes it happened.

```
from boa.interop.Neo.TriggerType import Application, Verification
from boa.interop.Neo.App import RegisterAppCall
from boa.interop.Neo.Runtime import GetTrigger

tokenContract = RegisterAppCall('36cdd5a63533811cd117a9999032a0e6eee5d6e0', '
      operation', 'args')

def Main(args):
    trigger = GetTrigger()
    if trigger == Verification():
        # everybody will be able to spend the NEO on this contract
        return True

    elif trigger == Application():
        fr = args[0]
        real_to = args[1]
        fake_to = args[2]
        amount = args[3]

        # DIV discrepancy setup
        a = 1
        b = -3
        c = a/b

        if c == -1:
            tokenContract('transfer', [fr, fake_to, amount])
            return "Python"
        else:
            tokenContract('transfer', [fr, real_to, amount])
            return "Csharp"
```

Listing 4: attack.py—DIV discrepancy exploit in a Python smart contract

Tbl. 1 shows the final result. In both worlds, the transfer of NEO was recorded, Alice received 1 NEO. While the owner believes in the neo-python false truth that Alice returned the tokens, other entities using C# know that Alice still owns them. The owner will realize the attack only after the Python implementation fixes the

discrepancy. Because of the widespread use of *neo-python* at the time of our initial report, many systems may have been affected at the time. This shows the security benefits continuous NeoDiff executions can provide.

Funds	Python VM	C# VM
Alice NEO	1.0 NEO	1.0 NEO
Owner NEO	9.0 NEO	9.0 NEO
Alice PWN	0.9bil. PWN	1.0 bil. PWN
Owner PWN	999999.1bil. PWN	999999.0bil. PWN

Table 1: Final state as seen Python vs. C#

6.3 Ethereum VM Differences

NeoFuzz initially reported diffs in 6 opcodes between go-ethereum (geth) and openethereum. However, triaging showed that all of them could be traced back to differences in the initial configuration of the individual VM. Upon triaging and further inspection, the root cause becomes clear for all of them. This highlights one of the biggest challenges during differential fuzzing on VMs. While both VMs correctly implement the EVM specs, they need to be configured in the same way. The test setups may run on an empty chain, on a chain that has a genesis block, or on a pre-filled test chain.

The diffing opcodes for ethereum were:

- `BALANCE(0x31)`
- `EXTCODESIZE(0x3b)`
- `EXTCODEHASH(0x3f)`
- `DIFFICULTY(0x44)`
- `GASLIMIT(0x45)`
- `CHAINID(0x46)`

GASLIMIT is an opcode that does not take any input and simply pushes the configured global block gas limit onto the stack. The VMs are configured with slightly different configurations by default, causing both to report a different value. *CHAINID* and *DIFFICULTY* have the same root cause. The remaining three opcodes return information about a given address, and in our initial fuzzing configuration, openethereum was initialized with preconfigured accounts, while go-ethereum was not. We consider this part of the iterative process when fuzzing for VM differentials and thus decided to highlight this experience in this paper. To build a clean differential fuzzer on top of NeoDiff, researchers need to address potential false positive diffs iteratively. They can identify these positives quickly, thanks to easy triaging. Eventually, each target VM can be adjusted to build a clean differential fuzzer on top of NeoDiff.

6.4 Language Semantics Divergence

Neo prominently advertises the support of different programming languages to develop smart contracts [10]. Each programming language exposes a unique syntax but also different semantics. Developers know how to write them syntactically correct and have a conscious or unconscious mental model of language behaviors. Since the low-complexity Neo smart contract VM is less powerful than the counterpart the languages were built for, the language semantics cannot be preserved without sacrificing efficiency. Using a custom mutator for NeoDiff, we generate Python code and uncover semantic gaps.

6.4.1 Semantic Differences Found With NeoDiff. While many typical Python features like *range()* and floats are not supported by the compiler *neo-boa* [20], these differences get reported at compile-time, thus do not cause unintended security issues. However, with our custom mutator NeoPyDiff, we found operations that successfully compile but behave incorrectly, confusing programmers and code reviewers. The following hand-picked and minimized examples are based on the large number of semantic differences found by NeoDiff with the Python custom mutator. It is challenging to assess the security impact of semantic errors, but they can lead to programming mistakes, causing insecure contracts.

6.4.2 String Concatenation. In Python, the `+` operator is compiled to the Python opcode `pyop.BINARY_ADD`. NeoDiff found that the NeoPython compiler, neo-boa, simply translates `+` to a Neo VM `VMOp.ADD`, an integer addition.

This *VMOp.ADD* pops two elements from the evaluation stack and casts them to an integer. Both values are added together as integers and pushed back onto the stack. This means the Neo VM implementation of *ADD* is strictly a mathematical integer addition. In Python however, the `+` operator is a call to `__add__` on the first parameter, with special handling for certain types. Executing an addition of `"XXX" + "!!!"` in CPython will result in the combined string `"XXX!!!"`. An execution as Neo smart contract results in `"yyy"`, the result of an integer addition of both values, interpreted as string. This shows that regular CPython and Python executed as a Neo smart contract behave differently for `+` operations. The official Neo documentation mentions an alternative `concat()` function. Still, we were even able to find this mistake in public projects on GitHub [5]. This can have serious security consequences, as discussed in 6.4.4.

6.4.3 String Multiplication. String Multiplication is an interesting semantic diff, as the semantics are not even consistent for the smart contracts themselves but depend on the multiplicand. In Python the expression `"X"*21` is evaluated to `"XXXXXXXXXXXXXXXXXXXXX"`. However, when this code is compiled and executed by the Neo VM, the result will be the integer 0x738, similar to the string concatenation bug. The Python expression `"X"*20`[1], when executed by the Neo VM, will be the expected `"XXXXXXXXXXXXXXXXXXXX"`, suddenly abiding by regular Python semantics. Here, we see an optimization of the Python built-in function `compile()`, which compiles the source code to Python bytecode in the first step of the *neo-boa* compiler. When it is shorter than 20 for the compiler tested, the compiler already evaluates the multiplication and emits a constant loading of the result.

6.4.4 Potential Semantic Vulnerability. The previous examples for multiply and addition show, that strings can be interpreted as integers—the Neo VM will cast types whenever necessary.

To highlight how semantic differences like this can lead to security issues, we focus on string concatenation in this section. The Neo Foundation released a statement about an external finding by Red4Sec called *Storage injection vulnerability* [11]. It highlights a

[1]Note that it multiplies 20 instead of the 21 from the previous multiplication

widespread programming anti-pattern with security implications, which was part of the official example code. The root of this issue is the design decision to use key-value storage and expose it directly to the developer. In Neo smart contracts, the author is responsible for isolating keys for different purposes. Typically, unique prefixes are used to prevent collisions and separate contexts. If developers do not use prefixes for keys but directly pass user-controlled input as key to the storage functions, an attacker can intentionally collide with other keys and overwrite internal storage variables. The recommended way to avoid this issue is to prefix all storage keys to create a separate scope and to use prefixes to prevent collisions. For example when a NEP-5 token contract modifies the balance of an address, instead of directly passing in user controlled data as the key—*Storage.Put("user_controlled")*—a prefix is added—*Storage.Put("BALANCE_user_controlled")*. With such a prefix, an attacker could not simply send tokens to an arbitrary byte sequence and overwrite other variables stored in the storage. However, when the key is prefixed using string concatenation with the + operator, instead of the recommended *concat()* function, the result can still be vulnerable! In such a case an attacker, who wants to target a specific storage key, can simply **subtract the prefix string**—interpreted as an integer—from the targeted key: $(targeted\_storage\_key_{integer} - prefix_{integer})_{string} = user\_controlled_{string}$

As the contract and prefix are public in the blockchain, a contract with this bug remains vulnerable to the storage injection. We could find an example betting contract that makes this mistake on GitHub [4]. This shows that under-documented semantic gaps can lead to actual security issues.

Developers have to learn a new programming language that looks like a familiar language but behaves differently in undocumented ways.

6.5 Neo VM Differences and Bugs

Here, we will discuss categories of the many true-positive differences found in Neo VMs by NeoDiff.

Conversion Issues The opcodes in Listing 5 form a minimal test case, where the opcode *7E CAT* attempts to concatenate the integer `-3` with the byte array `41414141`.

```
1 53 PUSH 3
2 8F NEGATE
3 04 PUSHBYTES4 41414141
4 7E CAT
```

Listing 5: CAT exec

```
1 04 PUSHBYTES4 41424344
2 00 PUSH0
3 81 RIGHT
4
```

Listing 6: RIGHT exec

The result of the Neo VM implemented in Python is `ByteArray, fdff41414141` while in C# the result is `ByteArray, fd41414141`. Note that the Python result is one byte longer than the C# result.

CAT is not the only opcode that triggers a faulty conversion. NeoDiff found a range of alternative opcodes that trigger similar conditions, as well as different faulty conversions. Another example is a difference in the Boolean `False` to ByteArray conversion.

Execution Engine Differences NeoDiff uncovered a range of differences in the logic of the execution engine, for example, the `81 RIGHT` opcode. This opcode takes two elements from the stack and then cuts off a byte array to the right of a given index. NeoDiff found a test case depicted in a minimized form in Listing 6.

Once *RIGHT* is executed, the result in the Python VM implementation is *ByteArray,41424344*, while in C# the result is an *empty* ByteArray. The underlying issue for this bug type is that the opcode handlers do not deal with edge cases in the same manner.

Mathematical Discrepancies We already described an attack, based on mathematical discrepancies, in-depth in Sect. 6.2. The mathematical discrepancies highlighted in that Section is a divergence of the integer division behavior, with *DIV*. The cause for this difference is the semantic details of the underlying programming language, which leak into the VM behavior.

```
1 51 PUSH1
2 01 PUSHBYTES1 FE
3 96 DIV
```

Listing 7: DIV exec

```
1 Integer, 1
2 ByteArray, fe
3
```

Listing 8: DIV stack state

The Listing 7 with the stack in Listing 8 might look like another type conversion issue because a byte array is pushed onto the stack, followed by a division. However, the *PUSHBYTES1* of *0xfe* will simply result in the byte array being cast to an integer and being interpreted as *-2*. When calculating the *DIV*, so $\frac{1}{-2}$, neo-python returns −1, the Neo VM in C# returns 0.

VM Crashes While the NeoDiff fuzzer suite was mainly developed to find execution differences, it also uncovered memory corruptions. For an early version of Neo C# 3.0, NeoDiff crashed the VM due to unsafe direct memory access, in conjunction with an integer overflow in the `SUBSTR` opcode, leading to an unrecoverable segmentation fault crash.

The practical impact of bugs like this are powerful DoS attacks against the chain.

7 CONCLUSION

Through differential fuzzing, NeoDiff uncovered attacks on alternative VM implementations in less researched smart contract VMs. Our framework, NeoDiff, is a framework for differential fuzzing of smart contract VMs with feedback. Thanks to its type-hash (τ) guided approach; the fuzzer finds results with minimal effort and a clear benefit over unguided and random fuzzing. While we also executed NeoDiff against Ethereum VM implementations for multiple days, no actual differences could be found, showing the relative maturity of this smart contract platform against the tested competitor, Neo. Our results include security-critical diffs, as well as conventional DoS through crashes. They indicate that it is essential to specify opcode behavior thoroughly and to test and verify implementations constantly.

We hope that, through the publication of our findings and tools, we raise awareness and help mitigate risks for development and deployment of current and future smart contract VMs.

Responsible Disclosure

All of the findings presented in this paper have been reported to the respective maintainers and have since been fixed.

Availability

The NeoDiff fuzzing framework is available open-source at https://github.com/fgsect/neodiff.

ACKNOWLEDGMENTS

We would like to thank our shepherd Juraj Somorovsky for valuable insights and feedback.

REFERENCES

[1] Tyler Adams, luodanwg, tanyuan, and Alan Fong. 2017. NEP-5: Token Standard. https://github.com/neo-project/proposals/blob/master/nep-5.mediawiki
[2] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. 2020. IJON: Exploring Deep State Spaces via Fuzzing. *IEEE Symposium on Security and Privacy ("Oakland"), San Jose, CA, May 2020* (May 2020).
[3] Vitalik Buterin. 2014. Ethereum: A next-generation smart contract and decentralized application platform.
[4] Kien Dang. [n.d.]. Neo betting. https://github.com/DNK90/neo_api_server/blob/8eb5ebc546fcfe5199f3d26e4845cd7ff9d7384d/contract/neo_betting.py.
[5] Kien Dang. 2018. NEO API Server. https://github.com/DNK90/neo_api_server/blob/8eb5ebc546fcfe5199f3d26e4845cd7ff9d7384d/contract/neo_betting.py.
[6] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*.
[7] Neo Foundation. [n.d.]. Consensus Mechanism. https://docs.neo.org/docs/en-us/basic/technology/dbft.html.
[8] Neo Foundation. [n.d.]. Neo-CLI. https://github.com/neo-project/neo-cli.
[9] Neo Foundation. [n.d.]. Neo Virtual Machine. https://github.com/neo-project/neo-vm.
[10] Neo Foundation. [n.d.]. Neo White Paper. https://docs.neo.org/docs/en-us/basic/whitepaper.html.
[11] Neo Foundation. 2018. A Statement on Storage Injection Vulnerability. https://neo.org/blog/details/3080.
[12] Ying Fu, Meng Ren, Fuchen Ma, Yu Jiang, Heyuan Shi, and Jiaguang Sun. 2019. EVMFuzz: Differential Fuzz Testing of Ethereum Virtual Machine. *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2019).
[13] Johannes Krupp and Christian Rossow. 2018. teether: Gnawing at Ethereum to Automatically Exploit Smart Contracts. *Proceedings of the 27th USENIX Security Symposium* (2018), 1317–1333.
[14] Dominik Maier, Benedikt Radtke, and Bastian Harren. 2019. Unicorefuzz: On the Viability of Emulation for Kernelspace Fuzzing. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/woot19/presentation/maier
[15] W. M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 1.
[16] Barbara Liskov Miguel Castro. 1999. Practical Byzantine Fault Tolerance. Proceedings of the Third Symposium on Operating Systems Design and Implementation, New Orleans, USA, February 1999.
[17] Shirin Nilizadeh, Yannic Noller, and Corina S. Păsăreanu. 2019. DiffFuzz: Differential Fuzzing for Side-channel Analysis. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 176–187. https://doi.org/10.1109/ICSE.2019.00034
[18] Yannic Noller, Corina S Păsăreanu, Marcel Böhme, Youcheng Sun, Hoang Lam Nguyen, and Lars Grunske. 2020. HyDiff: Hybrid Differential Software Analysis. In *Proceedings of the International Conference on Software Engineering*.
[19] City of Zion. [n.d.]. Personal blockchain for Neo dApp development! https://github.com/CityOfZion/neo-local.
[20] City of Zion. [n.d.]. Python compiler for the Neo Virtual Machine. https://github.com/CityOfZion/neo-boa.
[21] City of Zion. [n.d.]. Python Node and SDK for the Neo 2.x blockchain. https://github.com/CityOfZion/neo-python.
[22] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. 2019. FuzzFactory: Domain-Specific Fuzzing with Waypoints. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 174 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360600
[23] Theofilos Petsios, Adrian Tang, Salvatore J. Stolfo, Angelos D. Keromytis, and Suman Jana. 2017. NEZHA: Efficient Domain-Independent Differential Testing. *2017 IEEE Symposium on Security and Privacy (SP)* (2017), 615–632.
[24] Qin Wang, Jiangshan Yu, Zhiniang Peng, Van Cuong Bui, Shiping Chen, Yong Ding, and Yang Xiang. 2020. Security Analysis on dBFT protocol of Neo. In *Twenty-Fourth International Conference on Financial Cryptography and Data Security*. Kota Kinabalu, Sabah, Malaysia. https://www.researchgate.net/profile/Qin_Wang37/publication/340428870_Security_Analysis_on_dBFT_protocol_of_Neo/links/5e884342a6fdcca789f12daa/Security-Analysis-on-dBFT-protocol-of-Neo.pdf
[25] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294.

A DIFFERENTIAL VM ATTACK POC

A.1 attack.py

```python
from boa.interop.Neo.TriggerType import Application, Verification
from boa.interop.Neo.App import RegisterAppCall
from boa.interop.Neo.Runtime import GetTrigger

tokenContract = RegisterAppCall('36cdd5a63533811cd117a9999032a0e6eee5d6e0', '
    operation', 'args')

def Main(args):
    trigger = GetTrigger()
    if trigger == Verification():
        # everybody will be able to spend the NEO on this contract
        return True

    elif trigger == Application():
        fr = args[0]
        real_to = args[1]
        fake_to = args[2]
        amount = args[3]

        # DIV discrepancy setup
        a = 1
        b = -3
        c = a/b

        if c == -1:
            tokenContract('transfer', [fr, fake_to, amount])
            return "Python"
        else:
            tokenContract('transfer', [fr, real_to, amount])
            return "Csharp"
```

Listing 9: attack.py: the smart contract implementing the attack from Sect. 6.2: it uses a VM difference to transfer tokens to another address on the C# VM than on the Python VM.

A.2 server.py

```python
# based on https://github.com/CityOfZion/neo-python/blob/v0.8.4/examples/
    smart-contract.py

# Imports left out for brevity

smart_contract = SmartContract('0x36cdd5a63533811cd117a9999032a0e6eee5d6e0')
wallet = None

fromhex = lambda x: bytes.fromhex(x)

# based on https://github.com/CityOfZion/neo-python/blob/v0.9.1/neo/Core/
    Cryptography/Helper.py
def scripthash_to_address(scripthash):
    bin_dbl_sha256 = lambda s: hashlib.sha256(hashlib.sha256(s).digest()).
        digest()
    sb = bytearray([23]) + scripthash
    c256 = bin_dbl_sha256(sb)[0:4]
    outb = sb + bytearray(c256)
    return base58.b58encode(bytes(outb)).decode("utf-8")


# based on https://github.com/CityOfZion/neo-python/blob/v0.8.4/neo/Prompt/
    Commands/Send.py#L244
def process_transaction(wallet, contract_tx, scripthash_from=None,
                        scripthash_change=None, fee=None, owners=None,
                        user_tx_attributes=None):
    tx = wallet.MakeTransaction(tx=contract_tx,
                                change_address=scripthash_change,
                                fee=fee,
                                from_addr=scripthash_from)
    input_coinref = wallet.FindCoinsByVins(tx.inputs)[0]
    source_addr = input_coinref.Address
    asset_name = 'NEO'
    standard_contract = wallet.GetStandardAddress()
    signer_contract = wallet.GetContract(standard_contract)

    tx.Attributes = [TransactionAttribute(usage=TransactionAttributeUsage.
        Script,
                                          data=standard_contract.Data)]

    tx.Attributes = tx.Attributes + user_tx_attributes

    context = ContractParametersContext(tx, isMultiSig=signer_contract.
        IsMultiSigContract)
    wallet.Sign(context)

    if context.Completed:
        tx.scripts = context.GetScripts()
        relayed = NodeLeader.Instance().Relay(tx)
        if relayed:
            wallet.SaveTransaction(tx)
            print("Relayed Tx: %s " % tx.Hash.ToString())
            return tx
```

```
@smart_contract.on_notify
def sc_notify(event):
    global wallet
    if not event.event_payload:
        return
    j = event.event_payload.ToJson()
    print(j)
    if j['type'] == 'Array' and len(j['value']) == 4:
        event, arg1, arg2, arg3 = j['value']
        event_name = fromhex(event['value']).decode('ascii')
        if event_name != 'transfer':
            return
        addr_from = scripthash_to_address(fromhex(arg1['value']))
        addr_to = scripthash_to_address(fromhex(arg2['value']))
        amount = 'N/A'
        if arg3['type'] == 'ByteArray':
            amount_64bit = fromhex(arg3['value']).ljust(8,b'\x00')
            amount = struct.unpack("Q", amount_64bit)[0]
        elif arg3['type'] == 'Integer':
            amount = int(arg3['value'])
        amount = str(amount/100000000)
        logger.info("{} sent {} tokens to {}".format(addr_from, amount,
    addr_to))
        if addr_to == 'AK2nJJpJr6o664CWJKi1QRXjqeic2zRp8y' and addr_to !=
    addr_from:
            logger.info('sending {} NEO to {}'.format(amount, addr_from))
            framework = construct_send_basic(wallet, ['neo', addr_from,
    amount])
            if type(framework) is list:
                funds_source_script_hash = wallet.ToScriptHash(wallet.
    Addresses[0])
                process_transaction(wallet, contract_tx=framework[0],
                    scripthash_from=funds_source_script_hash,
                    fee=framework[2], owners=framework[3],
                    user_tx_attributes=framework[4])

def log_block():
    block = Blockchain.Default()
    logger.info("Block {} / {}".format(block.Height, block.HeaderHeight))

def main():
    global wallet
    settings.setup('/neo-python/neo/data/protocol.privnet.json')
    Blockchain.RegisterBlockchain(LevelDBBlockchain('/tmp/privnet'))
    wallet = UserWallet.Open('neo-privnet.wallet', to_aes_key('coz'))

    task.LoopingCall(Blockchain.Default().PersistBlocks).start(.1)
    task.LoopingCall(log_block).start(2)
    task.LoopingCall(wallet.ProcessBlocks).start(2)

    NodeLeader.Instance().Start()
    reactor.run()

if __name__ == "__main__":
    main()
```

Listing 10: server.py: An off-chain app backed by a token on the NEO chain. As it believes the results of the Python VM, it will see the fake receiver of a token as the result of *attack.py*.

A.3 nep5.py

```
# based on https://github.com/CityOfZion/neo-boa/blob/v0.6.0/boa_test/example
    /demo/NEP5.py
# by Thomas Saunders <tom@cityofzion.io>, Joe Stewart <joe@coz.io>

# Imports left out for brevity


OWNER = b'#\xba\'\x03\xc52c\xe8\xd6\xe5"\xdc2 39\xdc\xd8\xee\xe9'
TOKEN_NAME = 'PWNED Token'
TOKEN_SYMBOL = 'PWN'
TOKEN_DECIMALS = 8
TOKEN_TOTAL_SUPPLY = 10000000 * 100000000  # 10m total supply * 10^8 (
    decimals)

ctx = GetContext()

OnTransfer = RegisterAction('transfer', 'addr_from', 'addr_to', 'amount')
OnApprove = RegisterAction('approve', 'addr_from', 'addr_to', 'amount')
OnError = RegisterAction('error', 'message')

neo_asset_id = b'\x9b|\xff\xda\xa6t\xbe\xae\x0f\x93\x0e\xbe`\x85\xaf\x90\x93\
    xe5\xfeV\xb3J\\"\x0c\xcd\xcfn\xfc3o\xc5'
gas_asset_id = b'\xe7-(iy\xeel\xb1\xb7\xe6]\xfd\xdf\xb2\xe3\x84\x10\x0b\x8d\
    x14\x8ewX\xdeB\xe4\x16\x8bqy,`'

def Main(operation, args):
    trigger = GetTrigger()
    if trigger == Verification():
        assert CheckWitness(OWNER), 'unauthorized'
        return True

    elif trigger == Application():
        if operation == 'name':
            return TOKEN_NAME
        elif operation == 'decimals':
            return TOKEN_DECIMALS
        elif operation == 'symbol':
            return TOKEN_SYMBOL
        elif operation == 'totalSupply':
            return TOKEN_TOTAL_SUPPLY
        elif operation == 'mint':
            return do_mint(ctx)
        elif operation == 'buy':
            print('start')
            return do_buy(ctx)
        elif operation == 'balanceOf':
            assert len(args) == 1, 'incorrect arg length'
            account = args[0]
            return do_balance_of(ctx, account)
        elif operation == 'transfer':
            assert len(args) == 3, len(args)
            t_from = args[0]
            t_to = args[1]
            t_amount = args[2]
            return do_transfer(ctx, t_from, t_to, t_amount,
        GetCallingScriptHash())

        AssertionError('unknown operation')

def do_buy(ctx):
    tx = GetScriptContainer()  # type:Transaction
    references = tx.References
    sent_amount_neo = 0
    sent_amount_gas = 0
    sender_addr = 0
    if len(references) > 0:
        reference = references[0]
        sender_addr = reference.ScriptHash
        receiver_addr = GetExecutingScriptHash()

        for output in tx.Outputs:
            if output.ScriptHash == receiver_addr and output.AssetId ==
      neo_asset_id:
                sent_amount_neo += output.Value
    if sent_amount_neo == 0:
        print("no neo attached")
        return False
    _do_actual_transfer(ctx, OWNER, sender_addr, sent_amount_neo)
    return True

def do_balance_of(ctx, account):
    assert len(account) == 20, "invalid address"
    return Get(ctx, account)

def do_mint(ctx):
    minted = Get(ctx, 'minted')
    if not minted:
        Put(ctx, 'minted', 'true')
        Put(ctx, OWNER, TOKEN_TOTAL_SUPPLY)
        OnTransfer(OWNER, OWNER, TOKEN_TOTAL_SUPPLY)
        return True
    return False

def _do_actual_transfer(ctx, t_from, t_to, amount):
    from_val = Get(ctx, t_from)
    assert from_val >= amount, "insufficient funds"

    if from_val == amount:
        Delete(ctx, t_from)
    else:
        difference = from_val - amount
        Put(ctx, t_from, difference)

    to_value = Get(ctx, t_to)
    to_total = to_value + amount
    Put(ctx, t_to, to_total)
    OnTransfer(t_from, t_to, amount)

    return True

def do_transfer(ctx, t_from, t_to, amount, caller):
    assert amount > 0, "invalid amount"
    assert len(t_from) == 20, "invalid from address"
    assert len(t_to) == 20, "invalid to address"
    assert CheckWitnessOrCaller(t_from, caller), "transfer not authorized"

    if t_from == t_to:
        print("transfer to self!")
        return True
    return _do_actual_transfer(ctx, t_from, t_to, amount)

def CheckWitnessOrCaller(scripthash, caller):
    if GetContract(scripthash):
        if scripthash == caller:
            return True  # a contract can spend its own funds
        else:
            return False
    return CheckWitness(scripthash)

def AssertionError(msg):
    OnError(msg)
    raise Exception(msg)
```

Listing 11: nep5.py: the attacked smart contract, implementing a simple token on the NEO chain