Targeted Greybox Fuzzing with Static Lookahead Analysis

Valentin Wüstholz
ConsenSys Diligence/MythX, Germany
valentin.wustholz@consensys.net

Maria Christakis
MPI-SWS, Germany
maria@mpi-sws.org

ABSTRACT

Automatic test generation typically aims to generate inputs that explore new paths in the program under test in order to find bugs. Existing work has, therefore, focused on guiding the exploration toward program parts that are more likely to contain bugs by using an offline static analysis.

In this paper, we introduce a novel technique for targeted greybox fuzzing using an *online* static analysis that guides the fuzzer toward a set of *target locations*, for instance, located in recently modified parts of the program. This is achieved by first *semantically* analyzing each program path that is explored by an input in the fuzzer's test suite. The results of this analysis are then used to control the fuzzer's specialized power schedule, which determines how often to fuzz inputs from the test suite. We implemented our technique by extending a state-of-the-art, industrial fuzzer for Ethereum smart contracts and evaluate its effectiveness on 27 real-world benchmarks. Using an online analysis is particularly suitable for the domain of smart contracts since it does not require any code instrumentation—adding instrumentation to contracts changes their semantics. Our experiments show that targeted fuzzing significantly outperforms standard greybox fuzzing for reaching 83% of the challenging target locations (up to 14x of median speed-up).

ACM Reference Format:
Valentin Wüstholz and Maria Christakis. 2020. Targeted Greybox Fuzzing with Static Lookahead Analysis. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380388

1 INTRODUCTION

Automatic test generation is known to help find bugs and security vulnerabilities, and therefore, improve software quality. As a result, there has emerged a wide variety of test-generation tools that implement techniques such as random testing [24, 28, 67] and blackbox fuzzing [5, 8], greybox fuzzing [3, 6] as well as dynamic symbolic execution [18, 40] and whitebox fuzzing [17, 37, 41].

These techniques differ from each other in how much of the program structure they take into account. In general, the more structure a testing tool may leverage, the more effective it becomes in discovering new paths, but the less efficient it is in generating new inputs. For example, greybox fuzzing lies in the middle of this spectrum between performance and effectiveness in increasing coverage. In particular, it uses lightweight runtime monitoring that makes it possible to distinguish different paths, but it may not access any additional information about the program under test.

What these techniques have in common is that, just like any (static or dynamic) path-based program analysis, they can usually only explore a subset of all feasible paths in a program under test; for instance, in the presence of input-dependent loops. For this reason, path-based program analyses are typically not able to prove the absence of errors in a program, only their existence.

To make bug detection more effective, existing work has focused on guiding the exploration toward warnings reported by a static analysis (e.g., [29, 30, 38]), unverified program executions (e.g., [23, 35]), or sets of dangerous program locations (e.g., [14]). This is often achieved with an offline static analysis whose results are recorded and used to prune parts of the search space that is then explored by test generation.

The offline static analysis may be semantic, e.g., based on abstract interpretation, or not, e.g., based on the program text or its control-flow graph. A semantic analysis must consider all possible program inputs and states in which a piece of code may be executed. As a result, the analysis can quickly become imprecise, thus impeding its purpose of pruning as much of the search space as possible. For better results, one could resort to a more precise analysis, which would be less efficient, or to a more unsound analysis. The latter would limit the number of considered execution states in order to increase precision, but may also prune paths that are unsoundly verified [58].

Our approach. In this paper, we present a technique that *semantically* guides greybox fuzzing toward *target locations*, for instance, locations reported by another analysis or located in recently modified parts of the program. This is achieved with an *online* static analysis. In particular, the fuzzer invokes this online analysis right before adding a new input to its test suite. For the program path π that the new input explores (see bold path in Fig. 1), the goal of the analysis is to determine a path prefix π_{pre} for which all suffix paths are unable to reach a target location (e.g., T_x and T_y in Fig. 1). This additional information allows the fuzzer to allocate its resources more strategically such that more effort is spent on exercising program paths that might reach the target locations, thereby enabling *targeted fuzzing*. More precisely, this information feeds into a specialized power schedule of the fuzzer that determines how often to fuzz an input from the test suite.

We refer to our online static analysis as a *lookahead analysis* since, given a path prefix π_{pre}, it looks for reachable target locations along all suffix paths (sub-tree rooted at P_i in Fig. 1). We call the last program location of prefix π_{pre} a *split point* (P_i in Fig. 1). Unlike a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE '20, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380388

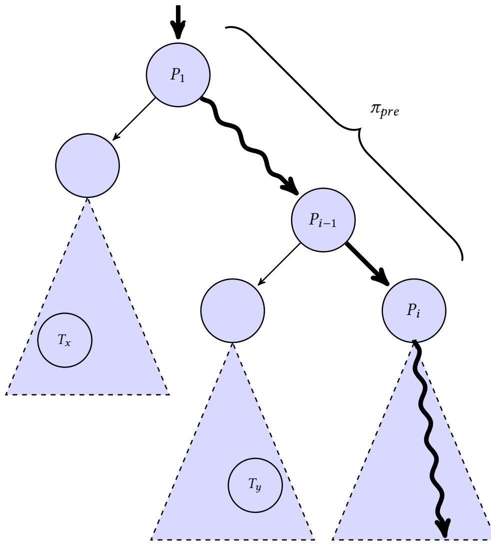

Figure 1: Execution tree of a program containing target locations T_x and T_y. The lookahead analysis analyzes a path π (bold) to identify a prefix π_{pre} such that no suffix paths reach a target location.

traditional static analysis, the lookahead analysis does not consider all possible execution states at the split point when analyzing all suffix paths—only the ones that are feasible along π_{pre}. In other words, the lookahead analysis combines the precision of a path-sensitive analysis along a feasible path prefix with the scalability of a path-insensitive suffix analysis. Intuitively, for a given path π, the precision of the lookahead analysis is determined by the number of suffix paths that are proved not to reach any target locations. Therefore, to optimize precision, the analysis tries to identify the *first* split point (P_i in Fig. 1) along π such that all targets are unreachable. Note that the lookahead analysis may consider any program location along π as a split point.

When combining greybox fuzzing with an online lookahead analysis, we faced four main challenges, which we address in this paper. In particular, we provide answers to the following questions: (1) How can the lookahead analysis effectively communicate its results to the fuzzer? (2) How lightweight can the analysis be to improve the effectiveness of the fuzzer in reaching target locations without having a negative impact on its performance? (3) How can the analysis be invoked from a certain split point along a path? (4) What are suitable split points for invoking the analysis to check all suffix paths?

Our implementation uses Harvey, a state-of-the-art, industrial greybox fuzzer for Ethereum smart contracts [79], which are programs managing crypto-currency accounts on a blockchain. We extended Harvey to incorporate Bran, a new static-analysis framework for smart contracts. A main reason for targeting the domain of smart contracts is that adding code instrumentation to contracts changes their semantics, and all existing techniques that use an offline static analysis rely on instrumenting the program under test. Our experiments on 27 benchmarks show that targeted fuzzing significantly outperforms standard greybox fuzzing for reaching 83% of the challenging target locations (up to 14x of median speed-up).

Contributions. We make the following contributions:

- We introduce a greybox-fuzzing algorithm that uses a lightweight, online static analysis and a specialized power schedule to guide the exploration toward target locations.
- We implement this fuzzing algorithm by extending the Harvey greybox fuzzer with Bran, a static analysis for smart contracts.
- We evaluate our technique on 27 real-world benchmarks and demonstrate that our lookahead analysis and power schedule significantly increase the effectiveness of greybox fuzzing in reaching target locations.

Outline. The next section provides background on greybox fuzzing and smart contracts. In Sect. 3, we give an overview of our technique through an example. Sect. 4 explains the technical details, and Sect. 5 describes our implementation. We present our experimental evaluation in Sect. 6, discuss related work in Sect. 7, and conclude in Sect. 8.

2 BACKGROUND

In this section, we review background on greybox fuzzing and smart contracts.

2.1 Greybox Fuzzing

Greybox fuzzing [3, 6] is a practical test-generation technique that has been shown to be very effective in detecting bugs and security vulnerabilities (e.g., [1]). Alg. 1 shows exactly how it works. (The grey boxes should be ignored.)

A greybox fuzzer takes as input the program under test *prog* and a set of seed inputs *S*. The fuzzer runs the program with the seeds (line 1) and associates each input with the unique identifier of the path it exercises, or *PID*. The *PIDs* data structure, therefore, represents a map from a *PID* to the corresponding input. Note that a path identifier is computed with lightweight runtime monitoring that allows the fuzzer to distinguish different program paths.

Next, the fuzzer selects an input from *PIDs* for mutation (line 3), which is typically performed randomly. This input is assigned an "energy" value, which indicates how long it should be fuzzed (line 5). The input is then mutated (line 8), and the program is run again with this new input (line 9). If the new input exercises a path that has not been seen before, it is added to *PIDs* with the corresponding path identifier (lines 10, 12).

This process terminates when a bound is reached, such as a timeout or a number of generated inputs (line 2). When that happens, the fuzzer returns a test suite comprising all inputs in *PIDs*, each exercising a different path in the program.

2.2 Smart Contracts

Ethereum [9] is one of the most well known blockchain-based [69, 72] computing platforms. Like a bank, Ethereum supports accounts that store a balance (in digital assets) and are owned by a user. More specifically, there is support for two account types, namely user and contract accounts.

Contract accounts are not managed by a user, but instead by a program. The program associated with a certain contract account describes an agreement between the account and any users that interact with it. For example, such a program could encode the rules of a gambling game. To store information, such as bets from various users, a contract account also comes with persistent state that the program may access and modify.

A contract account together with its managing program and persistent state is called a *smart contract*. However, the term may also refer to the code alone. Ethereum smart contracts can be developed in several high-level languages, such as Solidity and Vyper, which compile to Ethereum Virtual Machine (EVM) [78] bytecode.

Users interact with a smart contract, for instance to place a bet, by issuing a transaction with the contract. The transaction simply calls one of the contract functions, but in order to be carried out, users need to provide a fee. This fee is called *gas* and is approximately proportional to how much code needs to run. Any transaction that runs out of gas is aborted.

3 OVERVIEW

We now give an overview of our approach through the example of Fig. 2.

Example. The figure shows a constructed function `Bar`, which is written in Solidity and contained in a smart contract. (The comments should be ignored for now.) There are three assertions in this function, on lines 14, 19, and 22. A compiler will typically introduce a conditional jump for each assertion, where one branch leads to a location that fails. Let us assume that we select the failing locations (t_{14}, t_{19} , and t_{22}) of the three assertions as our target locations. Note that any target locations could be (automatically) selected based on various strategies, e.g., recently modified code (for smart contracts under development), assertions (added manually or by the compiler for checked errors such as division by zero), etc. Out of the above locations, t_{14} and t_{19} are unreachable, whereas t_{22} is reachable when input parameter `a` has value 42.

Generating a test input that reaches location t_{22} is difficult for a greybox fuzzer (and equally so for blackbox fuzzers) for two reasons. First, the probability of generating value 42 for parameter `a` is tiny, namely 1 out of 2^{256}. This means that, for the fuzzer to increase the chances of reaching t_{22}, it would need to fuzz certain "promising" inputs with a large amount of energy. However, standard greybox fuzzers are agnostic to what constitutes a promising input that is more likely to reach a target location when mutated.

Second, there are more than 100'000 program paths in function `Bar`. In fact, the then-branch of the first if-statement (line 5) contains two input-dependent loops (lines 11 and 16), whose number of iterations depends on parameters `w` and `z`, respectively. Recall that a greybox fuzzer generates new inputs by mutating existing ones from the test suite. Therefore, the larger the size of the test suite, the larger the space of possible mutations, and the lower the chances of generating an input that reaches the target location.

Existing work. As discussed earlier, there is existing work that leverages the results of an offline static analysis to guide automatic test generation toward unverified executions (e.g., [23, 29, 30, 35, 38]). To apply such a technique on the example of Fig. 2, let us assume a very lightweight static analysis that is based on abstract

```
1  function Bar(uint256 w, uint256 x, uint256 y,
2        uint256 z, uint256 a) returns (uint256)
3  {
4    uint256 ret = 0;
5    if (x % 2 == 0) {   // if (x % 1000 != 42) {
6      ret = 256;
7      if (y % 2 == 0) {
8        ret = 257;
9      }
10     w = w % ret;
11     while (w != 0) {
12       w--;
13     }
14     assert(w == 0);   // drop this line
15     z = z % ret;
16     while (ret != z) {
17       z++;
18     }
19     assert(ret == z); // assert(x != 42 - w*z);
20   } else {
21     ret = 3*a*a + 7*a + 101;
22     assert(ret != 5687);
23   }
24   return ret;
25 }
```

Figure 2: The running example.

interpretation [26, 27] and uses the simple constant-propagation domain [49]. Note that, for each program variable, the constant-propagation domain can only infer a single constant value. When run offline, this analysis is able to prove that target location t_{14} is unreachable. This is because, after the loop on line 11, the analysis assumes the negation of the loop condition (that is, `w == 0`), which is equivalent to the asserted condition.

However, the analysis cannot prove that location t_{19} is also unreachable. This is because, after the if-statement on line 7, variable `ret` has abstract value $\top$. In other words, the analysis finds `ret` to be unconstrained since the constant-propagation domain is not able to express that its value is either 256 or 257. Given that `ret` is $\top$, `z` remains $\top$ after the loop on line 16. It is, therefore, not possible for the analysis to determine whether these two variables always have the same value on line 19 and verify the assertion. As a result, automatic test generation needs to explore function `Bar` as if no static analysis had previously run. To check whether the assertion on line 19 always holds, a testing tool would have to generate inputs for all paths leading to it, thus including each iteration of the loop on line 11.

On the other hand, an existing technique for directed greybox fuzzing [14] performs lightweight instrumentation of the program under test to extract a distance metric for each input, which is then used as feedback for the fuzzer. So, the instrumentation encodes a static metric that measures the distance between the instrumented and the target locations in the control-flow graph. In our example, such metrics are less effective since all instructions are relatively close to the target locations, and the control-flow graph alone is not precise enough to determine more semantic reachability conditions.

In addition, when directly fuzzing bytecode or assembly, a control-flow graph might not be easily recoverable, for instance due to indirect jumps.

Lookahead analysis. In contrast, our approach alleviates the imprecision of a static analysis by running it online and does not require an explicit, complete control-flow graph. Our greybox fuzzer invokes the lookahead analysis for each input that is added to the test suite. Starting from split points (e.g., P_1, P_{i-1}, and P_i in Fig. 1) along an explored program path, the analysis computes a path prefix (π_{pre}) for which all suffix paths do not reach any target location (e.g., T_x and T_y). We refer to such a path prefix as a *no-target-ahead prefix* (see Def. 2 for more details). As we explain below, the lookahead analysis aims to identify short no-target-ahead prefixes.

As an example, let us consider the constant-propagation analysis and an input for function `Bar` with an even value for `x` (thus making execution take the then-branch of the first if-statement on line 5). Along the path exercised by this input, the analysis fails to show that both target locations t_{14} and t_{19} are unreachable for the suffix paths starting from line 7. In fact, the analysis is as imprecise as when run offline on the entire function. However, it does verify the unreachability of the target locations for all suffix paths from line 9 by propagating forward the constant value of variable `ret` (i.e., 257 for an even `y`, and 256 otherwise). Out of the many paths with an even value for `x`, the two no-target-ahead prefixes until line 9 (through the then- and else-branches of the if-statement on line 7) are actually the shortest ones for which the lookahead analysis proves that target locations t_{14} and t_{19} are unreachable.

Power schedule. The no-target-ahead prefixes computed by the lookahead analysis are used to control the fuzzer's power schedule [15], which assigns more energy to certain inputs according to two criteria.

First, it assigns more energy to inputs that exercise a rare (i.e., rarely explored) no-target-ahead prefix. The intuition is to fuzz these inputs more in order to increase the chances of flipping a branch along the rare prefix, and thereby, reaching a target location. Note that flipping a branch in a suffix path can never lead to a target location. For this reason, our power schedule no longer distinguishes inputs based on the program path they exercise, but rather based on their no-target-ahead prefix. To maximize the chances of discovering a target location with fuzzing, the lookahead analysis tries to identify the shortest no-target-ahead prefixes, which are shared by the most suffix paths.

For the example of Fig. 2, consider the two no-target-ahead prefixes (until line 9) that we discussed above. Consider also the no-target-ahead prefix until the successful branch of the assertion on line 22. The inputs that exercise these prefixes are dynamically assigned roughly the same energy by our schedule—if one of them is exercised more rarely than the others, it is given more energy. This makes reaching target location t_{22} significantly more likely than with standard power schedules based on path identifiers, which assign roughly the same energy to each input exercising one of the thousands of paths in `Bar`.

Second, our power schedule also assigns more energy to inputs exercising rare split points in a no-target-ahead prefix, similarly to how existing work assigns more energy to rare branches [54]. The intuition is the following. Any newly discovered no-target-ahead prefix is by definition rare—it has not been fuzzed before. Since it is rare, the power schedule will assign more energy to it, as discussed above. However, there are programs where new no-target-ahead prefixes can be easily discovered, for instance due to an input-dependent loop. In such cases, a power schedule only focusing on rare prefixes would prioritize these new prefixes at the expense of older ones that explore rare program locations, such as split points. For this reason, when a split point in a no-target-ahead prefix becomes rare, the power schedule tries to explore it more often.

As an example, consider the code in Fig. 2 while taking the comments into account, that is, replace lines 5 and 19 with the comments and drop line 14. The assertion on line 19 holds, but the constant-propagation analysis is too weak to prove it. As a result, for any path through this assertion, its no-target-ahead prefix has to include line 19. However, new no-target-ahead prefixes are very easily discovered; for instance, by exploring a different number of iterations in any of the two loops. So, even if at some point the fuzzer discovers the path that successfully exercises the assertion on line 22, its no-target-ahead prefix will quickly become less rare than any new prefixes going through the loops. The corresponding input will, therefore, be fuzzed less often even though it is very close to revealing the assertion violation. By prioritizing rare split points, for instance line 21, our power schedule will assign more energy to that input. This increases the chances of mutating the value of `a` to be 42 and reaching target t_{22}.

Both of these criteria effectively guide the fuzzer toward the target locations. For Fig. 2, our technique generates a test that reaches t_{22} in 27s on average (between 13 and 48s in 5 runs). Standard greybox fuzzing does not reach t_{22} in 4 out of 5 runs, with a timeout of 300s. The target location is reached in 113s during a fifth run, so in 263s on average. For this example, our technique achieves at least a 10x speed-up.

Why smart contracts. While our approach could in principle be applied to regular programs, it is particularly useful in the context of smart contracts. One reason is that, in this setting, combining an offline static analysis with test generation using code instrumentation would change the program semantics. Recall that a transaction with a smart contract is carried out when users provide enough gas, which is roughly proportional to how many instructions are run. Since instrumentation consumes gas at execution time, it could cause a test-generation tool to report spurious out-of-gas errors. Note that Harvey, the fuzzer we use in this work, does not instrument the program to collect path identifiers, but relies on call-backs from the virtual machine. Another reason for targeting smart contracts is that most deployed contracts are only available as bytecode, and recovering the control-flow graph from the bytecode is challenging.

4 TECHNIQUE

In this section, we describe our technique in detail by first formally defining a lookahead analysis (Sect. 4.1). We then discuss how to integrate such an analysis with greybox fuzzing to enable a more targeted exploration of the search space (Sect. 4.2). Lastly, we present a concrete algorithm for a lookahead analysis based on abstract interpretation.

4.1 Lookahead Analysis

Let us first define a prefix and a no-target-ahead prefix of a given path.

Definition 1 (**Prefix**). Given a program P and a path π in P, we say that π_{pre} is a *prefix* of π iff there exists a path ρ (which we call suffix) such that $\pi = concat(\pi_{pre}, \rho)$.

Note that, in the above definition, ρ may be empty, in which case $\pi = \pi_{pre}$.

Definition 2 (**No-target-ahead prefix**). Given a program P, target locations T, and a prefix π_{pre} of a path in P, we say that π_{pre} is a *no-target-ahead prefix* iff the suffix ρ of every path $\pi = concat(\pi_{pre}, \rho)$ in P does not contain a target location $\tau \in T$.

Note that any path π in a program P is trivially a no-target-ahead prefix since there cannot be any target locations after reaching the end of its execution.

For a given no-target-ahead prefix, the analysis computes a *lookahead identifier* (*LID*) that will later be used to guide the fuzzer.

Definition 3 (**Lookahead identifier**). Given a no-target-ahead prefix π_{pre}, the *lookahead identifier* λ is a cryptographic hash $hash(\pi_{pre})$.

The above definition ensures that it is very unlikely that two different no-target-ahead prefixes map to the same *LID*.

Unlike a path identifier (*PID*) in standard greybox fuzzing, which is computed purely syntactically, a *LID* captures a no-target-ahead prefix, which is computed by semantically analyzing a program path. As a result, two program paths with different *PIDs* may share the same *LID*. In other words, lookahead identifiers define equivalence classes of paths that share the same no-target-ahead prefix.

Definition 4 (**Lookahead analysis**). Given a program P, an input I, and a set of target locations T, a *lookahead analysis* computes a lookahead identifier λ for the corresponding no-target-ahead prefix π_{pre} (of path π exercised by input I) and a set of split points *SPs* along π_{pre}.

Note that a lookahead analysis that simply returns the hash of path π exercised by input I and all locations along π is trivially sound, but typically imprecise. For a given input, the precision of the analysis is determined by the length of the no-target-ahead prefix, and thereby, the number of suffix paths that are proved not to contain any target locations. In other words, the shorter the no-target-ahead prefix for a given input, the more precise the lookahead analysis.

4.2 Fuzzing with Lookahead Analysis

The integration of greybox fuzzing with a lookahead analysis builds on the following core idea. For each input in the test suite, the lookahead analysis determines a set of split points, that is, program locations along the explored path. It then computes a no-target-ahead prefix, which spans until one of these split points and is identified by a lookahead identifier. The fuzzer uses the rarity of the lookahead identifier as well as of the split points that are located along the no-target-ahead prefix to assign energy to the corresponding input.

The grey boxes in Alg. 1 highlight the key extensions we made to standard greybox fuzzing. For one, our algorithm invokes the lookahead analysis on line 11. This is done for every new input that is added to the test suite and computes the *LID* of the no-target-ahead prefix as well as the split points *SPs* along the prefix. Both are stored in the *PIDs* data structure for efficient lookups (e.g., when assigning energy).

Algorithm 1: Greybox fuzzing with lookahead analysis.

Input: Program *prog*, Seeds S, Target locations T

```
1  PIDs ← RunSeeds(S, prog)
2  while ¬Interrupted() do
3      input ← PickInput(PIDs)
4      energy ← 0
5      maxEnergy ← AssignEnergy(input)
6      maxEnergy ← LookaheadAssignEnergy(input)
7      while energy < maxEnergy do
8          input′ ← FuzzInput(input)
9          PID′ ← Run(input′, prog)
10         if IsNew(PID′, PIDs) then
11             LID, SPs ← LookaheadAnalyze(prog, input′, T)
12             PIDs ← Add(PID′, input′, LID, SPs, PIDs)
13         energy ← energy + 1
```

Output: Test suite Inputs(*PIDs*)

We also replace the existing power schedule on line 5 with a specialized one given by LookaheadAssignEnergy (line 6). As discussed in Sect. 3, our power schedule assigns more energy to inputs that exercise either a *rare LID* or a *rare split point* along a no-target-ahead prefix. We define the new power schedule in the following.

Definition 5 (**Rare *LID***). Given a test suite with *LIDs* Λ, a *LID* λ is rare iff

$$fuzz(\lambda) < rarity_cutoff,$$

where $fuzz(\lambda)$ measures the number of fuzzed inputs that exercised λ so far and $rarity_cutoff = 2^i$ such that

$$2^{i-1} < \min_{\lambda' \in \Lambda} fuzz(\lambda') \leq 2^i.$$

For example, if the *LID* with the fewest fuzzed inputs has been explored 42 times, then any *LID* that has been explored less than 2^6 times is rare.

The above definition is inspired by an existing power schedule for targeting rare branches [54] that introduced such a dynamically adjusted *rarity_cutoff*. Their experience shows that this metric performs better than simply considering the n *LIDs* with the lowest number of fuzzed inputs as rare.

Definition 6 (**Rare split point**). Given a test suite with split points *SPs* along the no-target-ahead prefixes, a split point p is rare iff

$$fuzz(p) < rarity_cutoff,$$

where $fuzz(p)$ measures the number of fuzzed inputs that exercised p so far and $rarity_cutoff = 2^i$ such that

$$2^{i-1} < \min_{p' \in SPs} fuzz(p') \leq 2^i.$$

Power schedule. Our power schedule is defined as follows for an input I with *LID* λ and split points *SPs* along the no-target-ahead

Algorithm 2: Lookahead algorithm.

Input: Program *prog*, Input *input*, Target locations T

```
1  π ← Run(input, prog)
2  i ← 0
3  SPs ← ∅
4  while i < |π| do
5      if IsSplitPoint(i, π) then
6          π_pre ← π[0..i + 1]
7          SPs ← SPs ∪ {π[i]}
8          φ, loc ← PrefixInference(π_pre)
9          if AreTargetsUnreachable(prog, loc, φ, T) then
10             return ComputeHash(π_pre), SPs
11     i ← i + 1
12 return ComputeHash(π), SPs
```

Output: Lookahead identifier λ, Split points *SPs*

prefix:

$$\begin{cases} \min(2^{selected(I)}, K), & \text{if } \lambda \text{ is rare} \lor \exists p \in SPs \cdot p \text{ is rare} \\ 1, & \text{otherwise.} \end{cases}$$

In the above definition, $selected(I)$ denotes the number of times that I was selected for fuzzing (line 3 in Alg. 1), and K is a constant (1024 in our implementation). Intuitively, our power schedule assigns little energy to inputs whose *LID* is not rare and whose no-target-ahead prefix does not contain any rare split points. Otherwise, it assigns much more energy, the amount of which depends on how often the input has been selected for fuzzing before. The energy grows exponentially up to some bound K, similarly to the cut-off-exponential schedule in AFLFast [15].

4.3 Lookahead Algorithm

Alg. 2 shows the algorithm for the lookahead analysis, which is implemented in function LookaheadAnalyze from Alg. 1 and uses abstract interpretation [26, 27].

First, the lookahead analysis executes the program input concretely to collect the exercised path π (line 1 in Alg. 2). Given path π, it searches for the shortest no-target-ahead prefix π_{pre} by iterating over possible split points p (lines 4–11). Let us explain these lines in detail.

On line 5, the algorithm calls a predicate IsSplitPoint, which is parametric in which locations constitute split points. All locations along π could be split points, but to narrow down the search, the implementation may consider only a subset of them, for instance, at conditional jumps.

At each split point, the analysis performs two separate steps: (1) *prefix inference* and (2) *suffix checking*. The prefix inference (line 8) statically analyzes the prefix π_{pre} using abstract interpretation to infer its postcondition ϕ. This step essentially executes the prefix in the abstract for all possible inputs that exercise this path.

Given condition ϕ, the analysis then performs the suffix checking to determine if all target locations are unreachable (line 9). This analysis performs standard, forward abstract interpretation by computing a fixed-point. If all target locations are unreachable, the analysis terminates and returns a non-empty *LID* by computing a hash over the program locations along the path prefix π_{pre} (line 10). This ensures that the analysis returns as soon as it reaches the first split point for which all targets are unreachable. In addition, it returns the set of all split points along prefix π_{pre}.

Even though off-the-shelf abstract interpreters are not designed to perform prefix inference and suffix checking, it is relatively straightforward to extend them. Essentially, when invoking a standard abstract interpreter on a program, the path prefix is always empty, whereas our lookahead analysis is partially path-sensitive (i.e., for the prefix, but not the suffix). Due to this partial path-sensitivity, even an inexpensive abstract domain (e.g., constant propagation or intervals) might be able to prove unreachability of a certain target location, which would otherwise require a more precise domain (for an empty prefix).

Split points. In practice, it is important to choose split points with care since too many split points will have a negative impact on the performance of the lookahead analysis. In our implementation, we only consider split points when entering a basic block for the first time along a given path. The intuition is that the lookahead analysis should run every time "new code" is discovered. Our experiments show that this design decision results in negligible overhead.

Calls. To keep the lookahead analysis lightweight, we analyze calls modularly. More specifically, any calls to other contracts are conservatively treated as potentially leading to target locations. Note that inter-contract calls are used very sparingly in smart contracts and that intra-contract calls are simply jumps.

5 IMPLEMENTATION

Our implementation extends Harvey [79]. It is actively used at one of the largest blockchain-security consulting companies[1] both for smart-contract audits and as part of an automated smart-contract analysis service[2] (more than 2.9M analyzed contracts from March to December 2019). For our purposes, we integrated Harvey with Bran, our new abstract-interpretation framework for EVM bytecode, which is open source[3].

Bran is designed to be scalable by performing a very lightweight, modular analysis that checks functional-correctness properties. Unlike other static analyzers for EVM bytecode (e.g., Securify [74] and MadMax [43]), Bran runs directly on the bytecode without having to reconstruct the control-flow graph or decompile to an intermediate language. Bran is equipped with a constant-propagation domain [49], which is commonly used in compiler optimizations. It handles all opcodes and integrates the go-ethereum virtual machine to concretely execute any opcodes with all-constant arguments.

Prefix length. During our preliminary experiments with the integration of Harvey and Bran, we observed that the prefix length may become quite large, for instance in the presence of input-dependent loops. However, the running time of the lookahead analysis is proportional to the prefix length, and our goal is to keep the analysis as lightweight as possible. For this reason, our implementation ignores any split points after the first 8'192 bytecode locations of the prefix. Note that this design decision does not affect the soundness of the lookahead analysis; it only reduces the search space of prefixes and might result in considering the entire path as the no-target-ahead prefix.

[1] https://consensys.net
[2] https://mythx.io
[3] https://github.com/Practical-Formal-Methods/bran

6 EXPERIMENTAL EVALUATION

We now evaluate our technique on real-world Ethereum smart contracts. First, we discuss the benchmark selection (Sect. 6.1) and describe our experimental setup (Sect. 6.2). We then evaluate the effectiveness of the static lookahead analysis in greybox fuzzing (Sect. 6.3) and identify potential threats to the validity of our experiments (Sect. 6.4).

6.1 Benchmark Selection

We evaluated our technique on a total of 27 smart contracts, which originate from 17 GitHub repositories. Tab. 1 gives an overview. The first column lists a benchmark identifier for each smart contract under test, while the second and last columns provide the name and description of the containing project. Note that a repository may contain more than one contract, for instance including libraries; from each repository, we selected one or more contracts for our evaluation. The third and fourth columns of the table show the number of public functions and lines of Solidity code in the benchmarks. (We provide links to all repositories as well as the changesets used for our experiments in a technical report [80].)

It is important to note that the majority of smart contracts are under 1'000 lines of code. Still, contracts of this size are complex programs, and each of them might take several weeks to audit. However, as it becomes clear from the example of Fig. 2, code size is not necessarily proportional to the number of feasible program paths or the difficulty to reach a particular target location with greybox fuzzing.

The repositories were selected with the goal of ensuring a diverse set of benchmarks. In particular, they include popular projects, such as the ENS domain name auction, the ConsenSys multisig wallet, and the MicroRaiden payment service. In addition to being widely known in the Ethereum community, these projects are highly starred on GitHub (5'136 stars in total on 2019-12-30, median 140), have been independently audited, and regularly transfer large amounts of assets. Moreover, our selection includes contracts from various application domains (like auctions, wallets, and tokens), attacked contracts (namely, The DAO and Parity wallet) as well as contracts submitted to the first Underhanded Solidity Coding Contest (USCC) [7]. Entries in this contest aim to conceal subtle vulnerabilities.

For selecting these repositories, we followed guidelines on how to evaluate fuzzers [51]. We do not randomly collect smart contracts from the Ethereum blockchain since this would likely contaminate our benchmarks with duplicates or bad-quality contracts—that is, contracts without users, assets, or dependencies, for instance, on libraries or other contracts.

6.2 Experimental Setup

Our experiments compare the integration of Harvey and Bran (incl. three variants) with Harvey alone to evaluate the effectiveness of targeted fuzzing. The comparison focuses on the time it takes for each configuration to cover a set of target locations. Harvey is the only greybox fuzzer for smart contracts, and there are no existing *targeted* black- or whitebox fuzzers for smart contracts. Implementing a targeted whitebox approach purely for comparison (for instance, based on symbolic execution such as Katch [63]) is beyond the scope of this paper. Existing work [14] already provides a detailed comparison showing how targeted grey- and whitebox approaches complement each other.

BIDs	Name	Functions	LoSC	Description
1	ENS	24	1205	ENS domain name auction
2–3	CMSW	49	503	ConsenSys multisig wallet
4–5	GMSW	49	704	Gnosis multisig wallet
6	BAT	23	191	BAT token (advertising)
7	CT	12	200	ConsenSys token library
8	ERCF	19	747	ERC Fund (investment fund)
9	FBT	34	385	FirstBlood token (e-sports)
10–13	HPN	173	3065	Havven payment network
14	MR	25	1053	MicroRaiden payment service
15	MT	38	437	MOD token (supply-chain)
16	PC	7	69	Payment channel
17–18	RNTS	49	749	Request Network token sale
19	DAO	23	783	The DAO organization
20	VT	18	242	Valid token (personal data)
21	USCC1	4	57	USCC'17 entry
22	USCC2	14	89	USCC'17 (honorable mention)
23	USCC3	21	535	USCC'17 (3rd place)
24	USCC4	7	164	USCC'17 (1st place)
25	USCC5	10	188	USCC'17 (2nd place)
26	PW	19	549	Parity multisig wallet
27	BNK	44	649	Bankera token
Total		662	12564	

Table 1: Overview of benchmarks. The first column lists a benchmark identifier for each smart contract under test, while the second and last columns provide the name and description of the containing project. The third and fourth columns provide the number of public functions and lines of source code in the benchmarks.

Targets. We randomly selected up to four target locations for each benchmark to avoid bias (e.g., by only targeting assertions or recently modified code). In particular, we picked contract locations of varying difficulty to reach, based on when they were first discovered during a 1h standard greybox-fuzzing run. So, we randomly picked at most one newly discovered location, if one existed, from each of the following time brackets in this order: 30–60m, 15–30m, 7.5–15m, 3.75–7.5m, and 1.875–3.75m. This ensures that all targets are reachable. Consequently, for a given prefix, any unreachable targets are proved so by the constant-propagation domain. This is possible mainly due to the path sensitivity provided by the prefix inference, which strengthens the capabilities of the imprecise constant-propagation domain during suffix checking.

Runs. We performed 24 runs of each configuration on the 27 benchmarks of Tab. 1. For each run, we used a different random seed, the same seed input, and a time limit of 1h (i.e., 3'600s). In our results, we report medians and use Wilcoxon-Mann-Whitney U tests to determine if differences in medians between configurations are statistically significant.

Machine. We used an Intel® Xeon® CPU @ 2.67GHz 24-core machine with 50GB of memory running Debian 9.5.

6.3 Results

We now evaluate the effectiveness of our technique by investigating five research questions.

RQ1: Effectiveness of targeted fuzzing. Tab. 2 compares our baseline configuration A, which does not enable the static lookahead analysis, with configuration B, which does. Note that configuration A uses the cut-off-exponential power schedule of AFLFast [15], whereas B uses our specialized schedule. The first two columns of the table indicate the benchmark and target IDs. Columns 3 and 4 show the median time (in seconds) required to discover the first input that reaches the target location (time-to-target) for both configurations, and column 5 shows the speed-up factor. Column 6 shows the p-value, which indicates the level of statistical significance; here, we use $p < 0.05$ for "significant" differences. The last two columns show Vargha-Delaney A12 effect sizes [75]. Intuitively, these measure the probability that configuration A is faster than B and vice versa.

For 32 (out of 60) target locations, we observe significant differences in time (i.e., $p < 0.05$), marked in bold in the table. *Configuration B significantly outperforms A for 31 (out of 32) of these target locations, with a median speed-up of up to 14x* for one of the targets in benchmark 26. In general, the results suggest that targeted fuzzing is very effective, and unsurprisingly, its impact is most significant for difficult targets (i.e., with high time-to-target for configuration A). Specifically, *for the 24 targets with $T_A \geq 900$ or $T_B \geq 900$, configuration B is significantly faster for 20, with insignificant differences between A and B for the remaining 4 targets.*

Note that running the static analysis with an empty prefix (resembling an offline analysis) on these benchmarks is not able to guide the fuzzer at all. Since all our target locations are reachable by construction, the analysis soundly reports them as reachable. Therefore, the fuzzer still needs to explore the entire contract to see if they indeed are.

RQ2: Effectiveness of lookahead analysis. To measure the effect of the lookahead analysis, we created configuration C, which is identical to configuration B except that the analysis is maximally imprecise and inexpensive. Specifically, AreTargetsUnreachable from Alg. 2 simply returns false, and consequently, the computed *LIDs* capture entire program paths, similarly to *PIDs*.

As shown in Tab. 3, there are significant differences between configurations B and C for 21 target locations. *Configuration B is significantly faster than C for 17 out of 21 targets*, and they are equally fast for 2 of the remaining 4 target locations.

Interestingly, configuration C is faster than A (for all 12 target locations with significant differences). This suggests that our power schedule regarding rare split points is effective independently of the lookahead analysis.

RQ3: Effectiveness of power schedule. To measure the effect of targeting rare *LIDs* and rare split points in our power schedule, we created configuration D. It is identical to configuration B except that it uses a variant of AFLFast's cut-off-exponential power schedule [15]. The original power schedule assigns energy to an input I based on how often its *PID* has been exercised. In contrast, our variant is based on how often its *LID* has been exercised and corresponds to using the results of the lookahead analysis with a standard power schedule.

However, as shown in Tab. 4, *configuration B is faster than configuration D for 28 of 30 targets (with significant differences).* This indicates that our power schedule significantly reduces the time-to-target, thus effectively guiding the fuzzer.

BID	Target ID	T_A	T_B	T_A/T_B	p	$A12_A$	$A12_B$
1	79145a51:35ee	324.15	**90.25**	3.59	0.049	0.33	0.67
1	79145a51:bd4	32.69	69.53	0.47	0.130	0.63	0.37
2	060a46c9:d03	3385.55	**706.71**	4.79	0.000	0.20	0.80
2	060a46c9:e29	161.66	106.57	1.52	0.197	0.39	0.61
2	060a46c9:16a5	701.39	**339.86**	2.06	0.008	0.27	0.73
2	060a46c9:1f11	346.06	**63.14**	5.48	0.000	0.11	0.89
3	708721b5:1485	396.11	394.54	1.00	0.477	0.44	0.56
3	708721b5:4ac	2292.00	**775.93**	2.95	0.000	0.19	0.81
3	708721b5:1ca0	1248.59	**817.76**	1.53	0.005	0.26	0.74
3	708721b5:1132	413.00	**216.72**	1.91	0.003	0.24	0.76
4	9b8e6b2a:d08	3600.00	**867.65**	4.15	0.000	0.15	0.85
4	9b8e6b2a:18f0	1657.33	**432.50**	3.83	0.002	0.24	0.76
4	9b8e6b2a:1fee	143.96	47.13	3.05	0.062	0.34	0.66
4	9b8e6b2a:553	3600.00	**833.70**	4.32	0.001	0.22	0.78
5	5a3e5a7f:c09	3600.00	**1282.42**	2.81	0.000	0.08	0.92
5	5a3e5a7f:23f	900.53	**466.99**	1.93	0.017	0.30	0.70
5	5a3e5a7f:1da8	1355.07	**646.41**	2.10	0.000	0.16	0.84
5	5a3e5a7f:1d67	1497.96	**524.08**	2.86	0.000	0.15	0.85
6	387bdf82:da7	61.66	22.70	2.72	0.089	0.36	0.64
8	e2aedada:15a7	2592.56	**1135.37**	2.28	0.002	0.24	0.76
8	e2aedada:17bb	1783.03	**612.39**	2.91	0.001	0.22	0.78
8	e2aedada:d71	73.93	47.89	1.54	0.307	0.41	0.59
8	e2aedada:13a8	258.14	**74.87**	3.45	0.035	0.32	0.68
9	dada6ee2:1693	334.82	**49.38**	6.78	0.000	0.13	0.87
9	dada6ee2:bee	225.12	**72.14**	3.12	0.000	0.19	0.81
9	dada6ee2:90e	84.62	50.39	1.68	0.338	0.42	0.58
10	d98d1d6b:1f10	1124.84	**281.45**	4.00	0.004	0.26	0.74
10	d98d1d6b:401a	164.12	153.95	1.07	0.861	0.48	0.52
10	d98d1d6b:3cdd	1669.91	1817.05	0.92	0.729	0.53	0.47
10	d98d1d6b:3ce8	3600.00	3600.00	1.00	0.713	0.47	0.53
11	3ae06fbe:34db	3600.00	3600.00	1.00	0.105	0.38	0.62
11	3ae06fbe:3de2	150.22	81.77	1.84	0.557	0.45	0.55
11	3ae06fbe:3ef3	284.34	395.15	0.72	0.703	0.47	0.53
11	3ae06fbe:10b2	238.35	142.03	1.68	0.228	0.40	0.60
12	0203d94d:713	76.82	60.27	1.27	0.910	0.49	0.51
14	b8c706d1:125e	3600.00	3600.00	1.00	0.085	0.39	0.61
14	b8c706d1:3479	290.73	299.26	0.97	0.861	0.52	0.48
14	b8c706d1:2023	34.65	43.72	0.79	0.992	0.50	0.50
15	06ef1a9c:27ce	3365.87	**467.90**	7.19	0.000	0.10	0.90
15	06ef1a9c:b41	100.00	73.83	1.35	0.877	0.49	0.51
15	06ef1a9c:a16	71.00	39.46	1.80	0.106	0.36	0.64
17	1c57401c:ef1	186.24	218.20	0.85	0.101	0.64	0.36
17	1c57401c:558	45.72	111.38	0.41	0.130	0.63	0.37
18	ac0bf5ee:15e4	1827.66	**321.36**	5.69	0.000	0.12	0.88
18	ac0bf5ee:171b	176.36	**48.04**	3.67	0.000	0.16	0.84
18	ac0bf5ee:15e0	133.84	**27.80**	4.81	0.001	0.22	0.78
18	ac0bf5ee:70c	**24.87**	61.47	0.40	0.036	0.68	0.32
20	54142e12:1555	29.57	15.42	1.92	0.298	0.41	0.59
23	d047b56e:5fb	42.01	20.70	2.03	0.279	0.41	0.59
24	b9ebdb99:40c	980.79	**139.78**	7.02	0.000	0.13	0.87
24	b9ebdb99:3d1	282.28	**57.21**	4.93	0.000	0.18	0.82
25	f1e90f8f:9fd	316.48	**24.61**	12.86	0.000	0.09	0.91
26	a788e7af:1f07	1778.07	**130.34**	13.64	0.000	0.07	0.93
26	a788e7af:1e29	2005.67	**336.04**	5.97	0.000	0.12	0.88
26	a788e7af:544	395.22	47.84	8.26	0.140	0.38	0.62
26	a788e7af:32b	44.67	45.92	0.97	0.813	0.48	0.52
27	9473c978:1541	2445.87	**324.46**	7.54	0.020	0.31	0.69
27	9473c978:e33	1493.03	**637.16**	2.34	0.023	0.31	0.69
27	9473c978:150e	178.11	97.60	1.82	0.120	0.37	0.63
27	9473c978:8e8	102.29	150.72	0.68	0.236	0.60	0.40

Table 2: Comparing time-to-target between configuration A (w/o lookahead analysis) and B (w/ lookahead analysis).

Nonetheless, configuration D is faster than A for all 6 targets with significant differences. This shows the effectiveness of the lookahead analysis independently of the power schedule.

BID	**Target ID**	$\mathbf{T_C}$	$\mathbf{T_B}$	$\mathbf{T_C/T_B}$	**p**	$\mathbf{A12_A}$	$\mathbf{A12_B}$
1	79145a51:35ee	223.45	90.25	2.48	0.718	0.47	0.53
1	79145a51:bd4	69.07	69.53	0.99	0.658	0.46	0.54
2	060a46c9:d03	2164.66	**706.71**	3.06	0.005	0.27	0.73
2	060a46c9:e29	156.18	106.57	1.47	0.338	0.42	0.58
2	060a46c9:16a5	854.32	**339.86**	2.51	0.042	0.33	0.67
2	060a46c9:1f11	56.01	63.14	0.89	0.926	0.49	0.51
3	708721b5:1485	527.02	394.54	1.34	0.797	0.48	0.52
3	708721b5:4ac	2000.32	**775.93**	2.58	0.007	0.27	0.73
3	708721b5:1ca0	775.97	817.76	0.95	0.327	0.42	0.58
3	708721b5:1132	298.71	216.72	1.38	0.317	0.41	0.59
4	9b8e6b2a:d08	3600.00	**867.65**	4.15	0.000	0.07	0.93
4	9b8e6b2a:18f0	1288.76	**432.50**	2.98	0.008	0.28	0.72
4	9b8e6b2a:1fee	88.80	47.13	1.88	0.557	0.45	0.55
4	9b8e6b2a:553	3508.27	**833.70**	4.21	0.000	0.10	0.90
5	5a3e5a7f:c09	3600.00	**1282.42**	2.81	0.000	0.09	0.91
5	5a3e5a7f:23f	2102.80	**466.99**	4.50	0.000	0.19	0.81
5	5a3e5a7f:1da8	1961.40	**646.41**	3.03	0.001	0.21	0.79
5	5a3e5a7f:1d67	1977.32	**524.08**	3.77	0.001	0.22	0.78
6	387bdf82:da7	20.35	22.70	0.90	0.317	0.59	0.41
8	e2aedada:15a7	2381.94	**1135.37**	2.10	0.004	0.26	0.74
8	e2aedada:17bb	915.16	612.39	1.49	0.071	0.35	0.65
8	e2aedada:d71	30.51	47.89	0.64	0.571	0.55	0.45
8	e2aedada:13a8	91.11	74.87	1.22	0.845	0.48	0.52
9	dada6ee2:1693	253.55	**49.38**	5.13	0.000	0.18	0.82
9	dada6ee2:bee	111.31	**72.14**	1.54	0.038	0.32	0.68
9	dada6ee2:90e	49.37	50.39	0.98	0.628	0.54	0.46
10	d98d1d6b:1f10	139.34	281.45	0.50	0.093	0.64	0.36
10	d98d1d6b:401a	145.53	153.95	0.95	0.829	0.52	0.48
10	d98d1d6b:3ce8	3600.00	3600.00	1.00	0.226	0.41	0.59
10	d98d1d6b:3cdd	3600.00	1817.05	1.98	0.146	0.38	0.62
11	3ae06fbe:34db	**3600.00**	**3600.00**	1.00	0.027	0.35	0.65
11	3ae06fbe:3de2	169.72	81.77	2.08	0.158	0.38	0.62
11	3ae06fbe:3ef3	182.77	395.15	0.46	0.135	0.63	0.37
11	3ae06fbe:10b2	214.12	142.03	1.51	0.942	0.51	0.49
12	0203d94d:713	44.13	60.27	0.73	0.516	0.56	0.44
14	b8c706d1:125e	**3600.00**	**3600.00**	1.00	0.010	0.35	0.65
14	b8c706d1:3479	108.30	299.26	0.36	0.110	0.64	0.36
14	b8c706d1:2023	40.71	43.72	0.93	0.845	0.52	0.48
15	06ef1a9c:27ce	2458.74	**467.90**	5.25	0.001	0.23	0.77
15	06ef1a9c:b41	59.20	73.83	0.80	0.228	0.60	0.40
15	06ef1a9c:a16	57.15	39.46	1.45	0.529	0.45	0.55
17	1c57401c:ef1	**104.23**	218.20	0.48	0.009	0.72	0.28
17	1c57401c:558	**63.84**	111.38	0.57	0.009	0.72	0.28
18	ac0bf5ee:15e4	719.04	**321.36**	2.24	0.007	0.27	0.73
18	ac0bf5ee:171b	106.78	**48.04**	2.22	0.002	0.23	0.77
18	ac0bf5ee:15e0	21.29	27.80	0.77	0.370	0.58	0.42
18	ac0bf5ee:70c	26.28	61.47	0.43	0.051	0.66	0.34
20	54142e12:1555	17.67	15.42	1.15	0.585	0.55	0.45
23	d047b56e:5fb	17.53	20.70	0.85	0.571	0.55	0.45
24	b9ebdb99:40c	178.49	139.78	1.28	0.138	0.37	0.63
24	b9ebdb99:3d1	115.03	57.21	2.01	0.089	0.36	0.64
25	f1e90f8f:9fd	114.00	**24.61**	4.63	0.000	0.16	0.84
26	a788e7af:1f07	323.97	130.34	2.49	0.089	0.36	0.64
26	a788e7af:1e29	404.19	336.04	1.20	0.797	0.48	0.52
26	a788e7af:544	142.41	47.84	2.98	0.464	0.44	0.56
26	a788e7af:32b	40.09	45.92	0.87	0.992	0.50	0.50
27	9473c978:1541	2320.70	324.46	7.15	0.210	0.39	0.61
27	9473c978:e33	1824.92	637.16	2.86	0.052	0.34	0.66
27	9473c978:150e	49.45	97.60	0.51	0.205	0.61	0.39
27	9473c978:8e8	95.71	150.72	0.63	0.244	0.60	0.40

Table 3: Comparing time-to-target for configurations B and C.

BID	**Target ID**	$\mathbf{T_D}$	$\mathbf{T_B}$	$\mathbf{T_D/T_B}$	**p**	$\mathbf{A12_A}$	$\mathbf{A12_B}$
1	79145a51:35ee	252.95	**90.25**	2.80	0.030	0.32	0.68
1	79145a51:bd4	64.12	69.53	0.92	0.688	0.53	0.47
2	060a46c9:d03	1734.13	**706.71**	2.45	0.013	0.29	0.71
2	060a46c9:e29	246.00	**106.57**	2.31	0.042	0.33	0.67
2	060a46c9:16a5	579.02	339.86	1.70	0.120	0.37	0.63
2	060a46c9:1f11	219.87	**63.14**	3.48	0.000	0.19	0.81
3	708721b5:1485	337.42	394.54	0.86	0.781	0.52	0.48
3	708721b5:4ac	1553.51	**775.93**	2.00	0.013	0.29	0.71
3	708721b5:1ca0	1001.05	817.76	1.22	0.183	0.39	0.61
3	708721b5:1132	353.12	**216.72**	1.63	0.049	0.33	0.67
4	9b8e6b2a:d08	1353.86	**867.65**	1.56	0.030	0.32	0.68
4	9b8e6b2a:18f0	1008.23	**432.50**	2.33	0.033	0.32	0.68
4	9b8e6b2a:1fee	172.58	**47.13**	3.66	0.002	0.24	0.76
4	9b8e6b2a:553	2464.13	**833.70**	2.96	0.000	0.16	0.84
5	5a3e5a7f:c09	3381.14	**1282.42**	2.64	0.001	0.21	0.79
5	5a3e5a7f:23f	515.76	466.99	1.10	0.220	0.40	0.60
5	5a3e5a7f:1da8	1197.92	**646.41**	1.85	0.002	0.24	0.76
5	5a3e5a7f:1d67	855.79	**524.08**	1.63	0.003	0.25	0.75
6	387bdf82:da7	110.41	**22.70**	4.86	0.000	0.18	0.82
8	e2aedada:15a7	2194.73	**1135.37**	1.93	0.002	0.24	0.76
8	e2aedada:17bb	1021.35	612.39	1.67	0.101	0.36	0.64
8	e2aedada:d71	82.30	47.89	1.72	0.097	0.36	0.64
8	e2aedada:13a8	188.01	74.87	2.51	0.051	0.34	0.66
9	dada6ee2:1693	279.31	**49.38**	5.66	0.001	0.23	0.77
9	dada6ee2:bee	195.79	**72.14**	2.71	0.006	0.27	0.73
9	dada6ee2:90e	45.93	50.39	0.91	0.992	0.50	0.50
10	d98d1d6b:1f10	606.63	281.45	2.16	0.085	0.35	0.65
10	d98d1d6b:3ce8	3600.00	3600.00	1.00	0.840	0.52	0.48
10	d98d1d6b:401a	254.15	153.95	1.65	0.228	0.40	0.60
10	d98d1d6b:3cdd	1956.69	1817.05	1.08	0.857	0.48	0.52
11	3ae06fbe:34db	3591.91	3600.00	1.00	0.885	0.51	0.49
11	3ae06fbe:3de2	181.38	81.77	2.22	0.130	0.37	0.63
11	3ae06fbe:3ef3	383.75	395.15	0.97	0.158	0.38	0.62
11	3ae06fbe:10b2	163.65	142.03	1.15	0.781	0.48	0.52
12	0203d94d:713	38.85	60.27	0.64	0.220	0.60	0.40
14	b8c706d1:125e	3600.00	3600.00	1.00	0.449	0.45	0.55
14	b8c706d1:3479	501.51	299.26	1.68	0.338	0.42	0.58
14	b8c706d1:2023	62.22	43.72	1.42	0.164	0.38	0.62
15	06ef1a9c:27ce	2514.11	**467.90**	5.37	0.000	0.10	0.90
15	06ef1a9c:b41	119.89	73.83	1.62	0.252	0.40	0.60
15	06ef1a9c:a16	102.73	**39.46**	2.60	0.020	0.30	0.70
17	1c57401c:ef1	**89.83**	218.20	0.41	0.025	0.69	0.31
17	1c57401c:558	66.72	111.38	0.60	0.184	0.61	0.39
18	ac0bf5ee:15e4	947.01	**321.36**	2.95	0.020	0.30	0.70
18	ac0bf5ee:171b	177.27	**48.04**	3.69	0.004	0.25	0.75
18	ac0bf5ee:15e0	72.29	27.80	2.60	0.071	0.35	0.65
18	ac0bf5ee:70c	**29.28**	61.47	0.48	0.021	0.69	0.31
20	54142e12:1555	24.46	15.42	1.59	0.516	0.44	0.56
23	d047b56e:5fb	36.38	20.70	1.76	0.348	0.42	0.58
24	b9ebdb99:40c	785.68	**139.78**	5.62	0.000	0.15	0.85
24	b9ebdb99:3d1	221.02	**57.21**	3.86	0.000	0.15	0.85
25	f1e90f8f:9fd	232.58	**24.61**	9.45	0.000	0.01	0.99
26	a788e7af:1f07	533.02	**130.34**	4.09	0.016	0.30	0.70
26	a788e7af:1e29	513.20	336.04	1.53	0.599	0.45	0.55
26	a788e7af:544	335.21	**47.84**	7.01	0.028	0.31	0.69
26	a788e7af:32b	72.62	45.92	1.58	0.543	0.45	0.55
27	9473c978:1541	1938.89	**324.46**	5.98	0.027	0.31	0.69
27	9473c978:e33	1517.21	**637.16**	2.38	0.024	0.31	0.69
27	9473c978:150e	160.07	97.60	1.64	0.093	0.36	0.64
27	9473c978:8e8	112.27	150.72	0.74	0.543	0.55	0.45

Table 4: Comparing time-to-target for configurations B and D.

RQ4: Scalability of lookahead analysis. One key design decision for using an *online* static analysis as part of a dynamic analysis (i.e., greybox fuzzing) was to make the static analysis as lightweight and scalable as sensible. That is why our lookahead analysis is modular and uses an inexpensive, and therefore scalable, constant-propagation domain.

Our results confirm that *the total running time of the lookahead analysis is a tiny fraction of the total running time of the fuzzer (0.09–105.93s of a total of 3600s per benchmark, median 2.73s).* This confirms that even a very lightweight static analysis can boost the effectiveness of fuzzing.

RQ5: Effect on instruction coverage. In our evaluation, *there were no noticeable instruction-coverage differences between any of our configurations.*

This indicates that our approach to targeted greybox fuzzing mainly affects the order in which different program locations are reached. Even though we prioritize certain inputs by assigning more energy to them, the fuzzer still mutates them randomly and eventually covers the same instructions as standard fuzzing. To avoid this, we would need to restrict some mutations (e.g., ones that never discover new *LIDs*), much like FairFuzz [54] restricts mutations that do not reach rare branches.

6.4 Threats to Validity

We have identified the following threats to validity.

External validity. A potential threat to the validity of our experiments has to do with external validity [71]. In particular, our results may not generalize to other contracts or programs. To alleviate this threat, we selected benchmarks from several, diverse application domains. Moreover, we provide the versions of all contracts used in our experiments so that others can also test them [80]. The results may also not generalize to other target locations, but we alleviate this threat by selecting them at random and with varying difficulty to reach.

Internal validity. Internal validity [71] is compromised when systematic errors are introduced in the experimental setup. A common pitfall in evaluating randomized approaches, such as fuzzing, is the potentially biased selection of seeds. During our experiments, when comparing the different configurations of our technique, we consistently used the same seed inputs for Harvey.

Construct validity. Construct validity ensures that any improvements, for instance in effectiveness or performance, achieved by a particular technique are due to that technique alone, and not due to other factors, such as better engineering. In our experiments, we compare different configurations of the same greybox fuzzer, and consequently, any effect on the results is exclusively caused by their differences.

7 RELATED WORK

Our technique for targeted greybox fuzzing leverages an online static analysis to semantically analyze each new path that is added to the fuzzer's test suite. The feedback collected by the static analysis is used to guide the fuzzer toward a set of target locations using a novel power schedule that takes inspiration from two existing ones [15, 54].

In contrast, the most closely related work [14] performs an offline instrumentation of the program under test encoding a static distance metric between the instrumented and the target locations in the control-flow graph. When running a given input, the instrumentation is used to obtain a dynamic (aggregated) distance. This distance subsequently guides the fuzzer toward the target locations.

Since a control-flow graph cannot always be easily recovered from EVM bytecode (e.g., due to indirect jumps), our lookahead analysis directly analyzes the bytecode using abstract interpretation [26, 27]. Our implementation uses the constant-propagation domain [49] to track the current state of the EVM (for instance, to resolve jump targets that are pushed to the execution stack). Unlike traditional static analyses, it aims to improve precision by performing a partially path-sensitive analysis—that is, path-sensitive for a prefix of a feasible path recorded at runtime by the fuzzer, and path-insensitive for all suffix paths.

Guiding greybox fuzzers. Besides AFLGo [14], there is a number of greybox fuzzers that target specific program locations [20], rare branches [54], uncovered branches [55, 76], or suspected vulnerabilities [22, 37, 46, 56]. While several of these fuzzers use an offline static analysis to guide the exploration, none of them leverages an online analysis.

Guiding other program analyzers. There is a large body of work on guiding analyzers toward specific target locations [60, 63] or potential failures [23, 29–32, 35, 38, 42, 61, 66] by combining static and dynamic analysis. These combinations typically perform an offline static analysis first and use it to improve the effectiveness of a subsequent dynamic analysis; for instance, by pruning parts of the program. For example, Check 'n' Crash [29] integrates the ESC/Java static checker [36] with the JCrasher test-generation tool [28]. Similarly, DyTa [38] combines the .NET static analyzer Clousot [33] with the dynamic symbolic execution engine Pex [73]. YOGI [42, 66] constantly refines its over- and under-approximations in the style of counterexample-driven refinement [25]. In contrast, our lookahead analysis is online and constitutes a core component of our targeted greybox fuzzer.

Hybrid concolic testing [62] combines random testing with concolic testing [18, 40, 70]. Even though the technique significantly differs from ours, it shares an interesting similarity: it uses online concolic testing during a concrete program execution to discover uncovered code on-the-fly. When successful, the inputs for covering the code are used to resume the concrete program execution.

Symbolic execution. In the context of symbolic execution [50], there have emerged numerous search strategies for guiding the exploration; for instance, to target deeper paths (in depth-first search), uncovered statements [68], or "less-traveled paths" [57]. Our technique resembles a search strategy in that it prioritizes exploration of certain inputs over others.

Compositional symbolic execution [11, 39] has been shown to be effective in merging different program paths by means of summaries in order to alleviate path explosion. Dynamic state merging [53] and veritesting [12] can also be seen as forms of summarization. Similarly, our technique merges different paths that share the same lookahead identifier for the purpose of assigning energy. The more precise the lookahead analysis, the shorter the no-target-ahead prefixes, and thus, the more effective the merging.

Program analysis for smart contracts. There is a growing number of program analyzers for smart contracts, ranging from random test generation frameworks [2, 47] to static analyzers and verifiers [4, 10, 13, 16, 19, 21, 34, 43–45, 48, 52, 59, 64, 65, 74, 77]. In contrast, we present a targeted greybox fuzzer for smart contracts, the first analyzer for contracts that incorporates static and dynamic analysis.

8 CONCLUSION

We have presented a novel technique for targeted fuzzing using static lookahead analysis. The key idea is to enable a symbiotic collaboration between the greybox fuzzer and an online static analysis. On one hand, dynamic information (i.e., feasible program paths) are used to boost the precision of the static analysis. On the other hand, static information about reachable target locations—more specifically, lookahead identifiers and split points—is used to guide the greybox fuzzer toward target locations. Our experiments on 27 real-world benchmarks show that targeted fuzzing significantly outperforms standard greybox fuzzing for reaching 83% of the challenging target locations (up to 14x of median speed-up).

We believe that the idea of using an online lookahead analysis to prune the search space of a path-based technique (e.g., fuzzing or symbolic execution) is not specific to smart contracts. However, the trade-offs (e.g., with respect to scalability of the lookahead analysis) may be significantly different in other settings (whether different programming languages, or different path-based techniques). We plan to investigate such variants in the future.

ACKNOWLEDGMENTS

We are grateful to the reviewers for their valuable feedback.

Maria Christakis's work was supported by DFG grant 389792660 as part of TRR 248 (see https://perspicuous-computing.science) and a Facebook Faculty Research Award.

REFERENCES

[1] [n.d.]. The AFL Vulnerability Trophy Case. http://lcamtuf.coredump.cx/afl/#bugs.
[2] [n.d.]. Echidna. https://github.com/trailofbits/echidna.
[3] [n.d.]. LibFuzzer—A Library for Coverage-Guided Fuzz Testing. https://llvm.org/docs/LibFuzzer.html.
[4] [n.d.]. Mythril. https://github.com/ConsenSys/mythril-classic.
[5] [n.d.]. Peach Fuzzer Platform. https://www.peach.tech/products/peach-fuzzer/peach-platform.
[6] [n.d.]. Technical "Whitepaper" for AFL. http://lcamtuf.coredump.cx/afl/technical_details.txt.
[7] [n.d.]. Underhanded Solidity Coding Contest. http://u.solidity.cc.
[8] [n.d.]. zzuf—Multi-Purpose Fuzzer. http://caca.zoy.org/wiki/zzuf.
[9] 2014. Ethereum White Paper. https://github.com/ethereum/wiki/wiki/White-Paper.
[10] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. 2018. Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL. In *CPP*. ACM, 66–77.
[11] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. 2008. Demand-Driven Compositional Symbolic Execution. In *TACAS (LNCS)*, Vol. 4963. Springer, 367–381.
[12] Thanassis Avgerinos, Alexander Rebert, Sang Kil Cha, and David Brumley. 2014. Enhancing Symbolic Execution with Veritesting. In *ICSE*. ACM, 1083–1094.
[13] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. 2016. Formal Verification of Smart Contracts: Short Paper. In *PLAS*. ACM, 91–96.
[14] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *CCS*. ACM, 2329–2344.
[15] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain. In *CCS*. ACM, 1032–1043.
[16] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, François Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A Scalable Security Analysis Framework for Smart Contracts. *CoRR* abs/1809.03981 (2018).
[17] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*. USENIX, 209–224.
[18] Cristian Cadar and Dawson R. Engler. 2005. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *SPIN (LNCS)*, Vol. 3639. Springer, 2–23.
[19] Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Yaron Velner. 2018. Quantitative Analysis of Smart Contracts. In *ESOP (LNCS)*, Vol. 10801. Springer, 739–767.
[20] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Grey-box Fuzzer. In *CCS*. ACM, 2095–2108.
[21] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-Optimized Smart Contracts Devour your Money. In *SANER*. IEEE Computer Society, 442–446.
[22] Animesh Basak Chowdhury, Raveendra Kumar Medicherla, and R. Venkatesh. 2019. VeriFuzz: Program Aware Fuzzing—(Competition Contribution). In *TACAS (LNCS)*, Vol. 11429. Springer, 244–249.
[23] Maria Christakis, Peter Müller, and Valentin Wüstholz. 2016. Guiding Dynamic Symbolic Execution Toward Unverified Program Executions. In *ICSE*. ACM, 144–155.
[24] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP*. ACM, 268–279.
[25] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-Guided Abstraction Refinement. In *CAV (LNCS)*, Vol. 1855. Springer, 154–169.
[26] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. ACM, 238–252.
[27] Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *POPL*. ACM, 269–282.
[28] Christoph Csallner and Yannis Smaragdakis. 2004. JCrasher: An Automatic Robustness Tester for Java. *SPE* 34 (2004), 1025–1050. Issue 11.
[29] Christoph Csallner and Yannis Smaragdakis. 2005. Check 'n' Crash: Combining Static Checking and Testing. In *ICSE*. ACM, 422–431.
[30] Mike Czech, Marie-Christine Jakobs, and Heike Wehrheim. 2015. Just Test What You Cannot Verify!. In *FASE (LNCS)*, Vol. 9033. Springer, 100–114.
[31] David Devecsery, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2018. Optimistic Hybrid Analysis: Accelerating Dynamic Analysis Through Predicated Static Analysis. In *ASPLOS*. ACM, 348–362.
[32] Matthew B. Dwyer and Rahul Purandare. 2007. Residual Dynamic Typestate Analysis Exploiting Static Analysis: Results to Reformulate and Reduce the Cost of Dynamic Analysis. In *ASE*. ACM, 124–133.
[33] Manuel Fähndrich and Francesco Logozzo. 2010. Static Contract Checking with Abstract Interpretation. In *FoVeOOS (LNCS)*, Vol. 6528. Springer, 10–30.
[34] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A Static Analysis Framework for Smart Contracts. In *WETSEB*. IEEE Computer Society/ACM, 8–15.
[35] Kostas Ferles, Valentin Wüstholz, Maria Christakis, and Isil Dillig. 2017. Failure-Directed Program Trimming. In *ESEC/FSE*. ACM, 174–185.
[36] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended Static Checking for Java. In *PLDI*. ACM, 234–245.
[37] Vijay Ganesh, Tim Leek, and Martin C. Rinard. 2009. Taint-Based Directed Whitebox Fuzzing. In *ICSE*. IEEE Computer Society, 474–484.
[38] Xi Ge, Kunal Taneja, Tao Xie, and Nikolai Tillmann. 2011. DyTa: Dynamic Symbolic Execution Guided with Static Verification Results. In *ICSE*. ACM, 992–994.
[39] Patrice Godefroid. 2007. Compositional Dynamic Test Generation. In *POPL*. ACM, 47–54.
[40] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *PLDI*. ACM, 213–223.
[41] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *NDSS*. The Internet Society, 151–166.
[42] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and SaiDeep Tetali. 2010. Compositional May-Must Program Analysis: Unleashing the Power of Alternation. In *POPL*. ACM, 43–56.
[43] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: Surviving Out-of-Gas Conditions in Ethereum Smart Contracts. *PACMPL* 2 (2018), 116:1–116:27. Issue OOPSLA.
[44] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2018. Online Detection of Effectively Callback Free Objects with Applications to Smart Contracts. *PACMPL* 2 (2018), 48:1–48:28. Issue POPL.
[45] Ákos Hajdu and Dejan Jovanovic. 2019. solc-verify: A Modular Verifier for Solidity Smart Contracts. In *VSTTE (LNCS)*. Springer. To appear.
[46] István Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *Security*. USENIX, 49–64.
[47] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *ASE*. ACM, 259–269.
[48] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *NDSS*. The Internet Society.
[49] Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *POPL*. ACM, 194–206.
[50] James C. King. 1976. Symbolic Execution and Program Testing. *CACM* 19 (1976), 385–394. Issue 7.

[51] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *CCS*. ACM, 2123–2138.
[52] Johannes Krupp and Christian Rossow. 2018. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *Security*. USENIX, 1317–1333.
[53] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient State Merging in Symbolic Execution. In *PLDI*. ACM, 193–204.
[54] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *ASE*. ACM, 475–485.
[55] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-State Based Binary Fuzzing. In *ESEC/FSE*. ACM, 627–637.
[56] Yuwei Li, Shouling Ji, Chenyang Lv, Yuan Chen, Jianhai Chen, Qinchen Gu, and Chunming Wu. 2019. V-Fuzz: Vulnerability-Oriented Evolutionary Fuzzing. *CoRR* abs/1901.01142 (2019).
[57] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. 2013. Steering Symbolic Execution to Less Traveled Paths. In *OOPSLA*. ACM, 19–32.
[58] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In Defense of Soundiness: A Manifesto. *CACM* 58 (2015), 44–46. Issue 2.
[59] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *CCS*. ACM, 254–269.
[60] Kin-Keung Ma, Yit Phang Khoo, Jeffrey S. Foster, and Michael Hicks. 2011. Directed Symbolic Execution. In *SAS (LNCS)*, Vol. 6887. Springer, 95–111.
[61] Lei Ma, Cyrille Artho, Cheng Zhang, Hiroyuki Sato, Johannes Gmeiner, and Rudolf Ramler. 2015. GRT: Program-Analysis-Guided Random Testing. In *ASE*. IEEE Computer Society, 212–223.
[62] Rupak Majumdar and Koushik Sen. 2007. Hybrid Concolic Testing. In *ICSE*. IEEE Computer Society, 416–426.
[63] Paul Dan Marinescu and Cristian Cadar. 2013. KATCH: High-Coverage Testing of Software Patches. In *ESEC/FSE*. ACM, 235–245.
[64] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. *CoRR* abs/1907.03890 (2019).
[65] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the Greedy, Prodigal, and Suicidal Contracts at Scale. (2018), 653–663.
[66] Aditya V. Nori, Sriram K. Rajamani, Saideep Tetali, and Aditya V. Thakur. 2009. The YOGI Project: Software Property Checking via Static Analysis and Testing. In *TACAS (LNCS)*, Vol. 5505. Springer, 178–181.
[67] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *ICSE*. IEEE Computer Society, 75–84.
[68] Sangmin Park, B. M. Mainul Hossain, Ishtiaque Hussain, Christoph Csallner, Mark Grechanik, Kunal Taneja, Chen Fu, and Qing Xie. 2012. CarFast: Achieving Higher Statement Coverage Faster. In *FSE*. ACM, 35.
[69] Siraj Raval. 2016. *Decentralized Applications: Harnessing Bitcoin's Blockchain Technology*. O'Reilly Media.
[70] Koushik Sen and Gul Agha. 2006. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In *CAV (LNCS)*, Vol. 4144. Springer, 419–423.
[71] Janet Siegmund, Norbert Siegmund, and Sven Apel. 2015. Views on Internal and External Validity in Empirical Software Engineering. In *ICSE*. IEEE Computer Society, 9–19.
[72] Melanie Swan. 2015. *Blockchain: Blueprint for a New Economy*. O'Reilly Media.
[73] Nikolai Tillmann and Jonathan de Halleux. 2008. Pex—White Box Test Generation for .NET. In *TAP (LNCS)*, Vol. 4966. Springer, 134–153.
[74] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *CCS*. ACM, 67–82.
[75] András Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *JEBS* 25 (2000), 101–132. Issue 2.
[76] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang Sun. 2018. SAFL: Increasing and Accelerating Testing Coverage with Symbolic Execution and Guided Fuzzing. In *ICSE Companion*. ACM, 61–64.
[77] Yuepeng Wang, Shuvendu K. Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, and Immad Naseer. 2019. Formal Verification of Workflow Policies for Smart Contracts in Azure Blockchain. In *VSTTE (LNCS)*. Springer. To appear.
[78] Gavin Wood. 2014. Ethereum: A Secure Decentralised Generalised Transaction Ledger. http://gavwood.com/paper.pdf.
[79] Valentin Wüstholz and Maria Christakis. 2019. Harvey: A Greybox Fuzzer for Smart Contracts. *CoRR* abs/1905.06944 (2019).
[80] Valentin Wüstholz and Maria Christakis. 2019. Targeted Greybox Fuzzing with Static Lookahead Analysis. *CoRR* abs/1905.07147 (2019).