Abstract—We present grey-box concolic testing, a novel path-based test case generation method that combines the best of both white-box and grey-box fuzzing. At a high level, our technique systematically explores execution paths of a program under test as in white-box fuzzing, a.k.a. concolic testing, while not giving up the simplicity of grey-box fuzzing: it only uses a lightweight instrumentation, and it does not rely on an SMT solver. We implemented our technique in a system called Eclipser, and compared it to the state-of-the-art grey-box fuzzers (including AFLFast, LAF-intel, Steelix, and VUzzer) as well as a symbolic executor (KLEE). In our experiments, we achieved higher code coverage and found more bugs than the other tools.

Index Terms—software testing, concolic testing, fuzzing

I. INTRODUCTION

Fuzz testing (fuzzing for short) has been the de facto standard for finding security vulnerabilities in closed binary code [1]. Security practitioners appreciate fuzzing because it always finds bugs along with proof. Major software companies such as Microsoft and Google employ fuzzing nowadays in their software development life cycle as a means of assuring the security of their products [2], [3].

Most notably, grey-box fuzzers such as AFL [4], AFLFast [5], Steelix [6], VUzzer [7], Angora [8], CollAFL [9], and T-Fuzz [10] are emerging as the state-of-the-art in bug finding. Grey-box fuzzing generates test cases with an evolutionary process. Specifically, it executes test cases and evaluates them based on a fitness function (a.k.a. an objective function). It then prioritizes those with better fitness, evolves them to find test cases that meet the objective, and continues to iterate the entire process with the hope of exercising a buggy path that triggers program crashes.

Current grey-box fuzzers use code coverage as their fitness function. Accordingly, they are sometimes referred to as coverage-based fuzzers [5], [7]. For example, AFL [4] and its successors [5], [6], [11] employ an approximated form of branch coverage, while VUzzer [7] uses weighted basic block hit counts as its fitness function. It is plain that the likelihood of exercising interesting execution paths of the Program Under Test (PUT) increases by maximizing the code coverage.

However, existing grey-box fuzzers suffer from exercising new branches even with the coverage-based guidance, as code coverage does not change sensitively over input mutations. In particular, two program executions with two different inputs may achieve the same code coverage, even though the compared values of a conditional branch in the executions are distinct. In other words, code coverage can provide feedback only if a conditional branch is penetrated with a randomly generated input, but it does not directly help generate such input. This lack of sensitivity makes it difficult for grey-box fuzzers to generate high-coverage test cases in some circumstances, for example when the PUT compares input to a specific magic value. Even the current state-of-the-art grey-box fuzzers such as AFLGo [11], Steelix [6] and VUzzer [7] have more or less the same problem.

Consequently, it is widely believed that grey-box fuzzing cannot be a sole test case generation algorithm despite its effectiveness at finding vulnerabilities. Therefore, grey-box fuzzers are often augmented by heavy-cost white-box analyses such as dynamic symbolic execution [10], [12] and fine-grained taint analyses [7], [8], [13], or by providing initial seed inputs to direct the test case generation process [14], [15]. For example, Angora [8] and Driller [12] leverage fine-grained taint analysis and dynamic symbolic execution, respectively, to improve code coverage of grey-box fuzzing.

Meanwhile, white-box fuzzing (a.k.a. dynamic symbolic execution or concolic testing) [16]–[21] can systematically generate test cases by solving branch conditions, but it is fundamentally limited by the scalability, leaving aside the classic path explosion problem. First, white-box fuzzers analyze every single instruction of the PUT. Because it instruments every single instruction of the PUT, every fuzzing iteration entails a significant computational cost. Second, symbolic execution builds up symbolic path constraints for every execution path. Solving such constraints with an SMT solver [22] is computationally expensive. Furthermore, storing symbolic expressions for every single memory cell affected by symbolic inputs requires significant memory space.

In this paper, we propose a novel test case generation technique, called grey-box concolic testing, and implement it in a tool referred to here as Eclipser. Grey-box concolic testing efficiently generates test cases satisfying branch conditions as in white-box fuzzing, while not losing simplicity: it does not rely on expensive program analysis techniques. Thus, it scales to real-world applications as in grey-box fuzzing.

Our approach resembles generational search, which is a search strategy widely used in white-box fuzzing [19], [23], where a single program execution produces a generation of test cases by resolving every conditional branch encountered during the execution. Grey-box concolic testing performs a path-based test case generation too, but it tries to resolve conditional branches in a grey-box manner: it instruments the
The key difference between grey-box concolic testing and white-box fuzzing is that our approach relies on an approximated form of path constraint, which partially describes input conditions to exercise each execution path of the PUT. The approximated path constraints help us find inputs that can penetrate conditional branches without resorting to CPU- or memory-intensive operations such as SMT solving. Naturally, the path constraints generated from grey-box concolic testing are imprecise, but, in practice, they are precise enough to quickly explore diverse execution paths. The primary design decision here is to trade off simplicity for precision.

Of course, the lack of precision introduces incomplete exploration of paths in the PUT, but Eclipser compensates this by alternating between grey-box concolic testing and classic grey-box fuzzing as in Driller [12]. Even though grey-box concolic testing does not fully cover conditional branches of the PUT, the grey-box fuzzing module continues to cover new paths and branches, and vice versa. We found that in practice this design decision effectively expands the capability of Eclipser beyond that of the current state-of-the-art grey- and white-box fuzzers in terms of both finding vulnerabilities and reaching high code coverage.

We evaluated Eclipser against current state-of-the-art fuzzers. The practicality of our system as a test case generator was confirmed by an experiment we performed against KLEE, a state-of-the-art symbolic executor known to excel in generating tests with high coverage from given source code [18], [24]. In the experiment, Eclipser achieved 8.57% higher code coverage than KLEE on GNU coreutils, which is a well-known benchmark used for evaluating test case generation algorithms [25]–[27], without the help of SMT solvers.

To evaluate Eclipser as a bug finding tool, we compared Eclipser against several state-of-the-art grey-box fuzzers such as AFLFast [5], LAF-intel [28], Steelix [6], and VUzzer [7]. We also ran Eclipser on 22 binaries extracted from Debian 9.1, and found 40 unique bugs from 17 programs. We have reported all the bugs we found to the developers. In summary, this paper has the following contributions.

1) We introduce a novel path-based test case generation algorithm, called grey-box concolic testing, which leverages lightweight instrumentation to generate high-coverage test cases.
2) We implement Eclipser and evaluate it on various benchmarks against state-of-the-art fuzzers including AFLFast, LAF-intel, Steelix, and VUzzer. According to the evaluation, Eclipser excels in terms of both code coverage and bug finding compared to them.
3) We ran Eclipser on 22 real-world Linux applications and found 40 previously unknown bugs. CVE identifiers were assigned for 8 of them.
4) We make the source code of Eclipser public for open science: https://github.com/SoftSec-KAIST/Eclipser.

II. BACKGROUND AND MOTIVATION

A. Grey-box Fuzzing

Fuzzing is essentially a process of repeatedly executing a Program Under Test (PUT) with generated test cases. Grey-box fuzzing [4]–[6], [29] evolves test cases within a feedback loop, in which executions of the PUT with each test case are evaluated by a criterion that we call a fitness function. Most grey-box fuzzers use code coverage as their fitness function, although specific implementations may differ. AFL [4], for instance, uses branch coverage (modulo some noise) to determine which input should be fuzzed next.

Despite their recent success, coverage-based grey-box fuzzers are linked to a major drawback in that their fuzzing process involves too many unnecessary trials to find a test case that exercises a specific branch. This is mainly due to the insensitivity of the fitness function used for fuzzing. Informally speaking, a fitness function is sensitive if the fitness can be varied easily by a small modification of the input value. Any code coverage metric, e.g., node coverage and branch coverage, is insensitive because there is no intermediate fitness between two executions that cover the true and the false branch. Therefore, it is difficult to find an input that flips a given branch condition.

The necessity of sensitive fitness function is widely recognized in search-based software testing [30] where test case generation is considered as an optimization problem. One notable fitness function is branch distance [31], [32], which is a distance between the operand values of a conditional branch. Fuzzing community has been recently started to employ the idea: Angora [8] leveraged branch distance to improve its fuzzing performance. Eclipser leverages the similar insight, but uses the sensitivity to directly infer and solve approximated branch conditions, not leaning on metaheuristics. Both approaches are orthogonal and complementary to each other.

B. Notation and Terminologies

We let an execution be a finite sequence of instructions: we do not consider a program execution with an infinite loop for instance. This is not an issue in fuzzing, because fuzzers will forcefully terminate the PUT after a certain period of time, which is typically a parameter to fuzzers. We denote an execution of a program \( p \) with an input \( i \) by \( \sigma_p(i) \). In our model, an input is a byte sequence, although we can easily extend it to represent a bit string. For a given input \( i \), we let \( i[n] \) be the \( n \)th byte value of \( i \). We denote an input derived by modifying \( i[n] \) to become \( v \) by \( i[n] \leftarrow v \). Throughout the paper, we interchangeably use the terms test case and test input. We let an input field be a consecutive subsequence of an input. There can be many input fields for a given input, and input fields may overlap.

Approximate Path Constraint. In symbolic execution [19], a path constraint is a predicate on the input such that if an execution path is feasible, then the corresponding path condition is satisfiable. Since our approach tries to be lightweight, we do not trace the exact path conditions, but an approximated version that we call an approximate path constraint.
int vulnfunc(int32_t intInput, char * strInput) {
    if (2 * intInput + 1 == -31337)
        return crash();
    if (strcmp(strInput, "Bad!") == 0)
        return 0;
    int fd = open(argv[1], O_RDONLY);
    read(fd, buf, sizeof(buf) - 1);
    buf[8] = 0;
    vulnfunc((int32_t *)&buf[0], &buf[4]);
    return 0;
}

(a) An example program written in C. Error handling routines are intentionally not shown for simplicity.

(b) Comparison between state-of-the-art fuzzers in our example program.

Fig. 1. Our motivating example and a comparison of different fuzzers.

Seed. In this paper, we let seed be a data structure that represents an input for a specific program. We denote a seed for a program p as $s_p$, and the execution of p with the seed $s_p$ as $\sigma_p(s_p)$. The nth byte of the seed $s_p$ is denoted by $s_p[n]$. Every byte of a seed is tagged with a field “constr”, which is an independent subset of an approximate path constraint with regard to the byte. We can access an approximate path constraint of the nth byte of a seed $s_p$ with the dot notation: $s_p[n]$.constr. For a given seed $s_p$, the nth byte of the seed $s_p[n]$ should satisfy $s_p[n]$.constr in order to exercise the same execution path as $\sigma_p(s_p)$.

C. Motivation

Figure 1a shows an example program that motivates our research. Note that we use C representation for ease of explanation, although our system works on raw binary executables. It takes in a file as input, and uses the first 4 bytes of the file as an integer, and the rest 4 bytes as a 5-byte string by appending a NULL character at the end (Line 10). These two values are used as parameters to the function vulnfunc. In order to find the crash in Line 4, we need to provide the 32-bit integer 15,668 and the string "Bad!" as input to the function.

Can current grey-box fuzzers find the test input that triggers this crash? How effective are grey-box fuzzers at finding such a simple bug? To answer these questions, we fuzzed our example program with 6 state-of-the-art fuzzers as well as with Eclipser for 1 hour each on a single core of Intel Xeon E3-1231 v3 processor (3.40 GHz). We selected four open-sourced grey-box fuzzers including AFL [4], AFLFast [5], AFLGo [11], and LAF-intel1 [28]. We also chose a popular symbolic executor, i.e., a white-box fuzzer, KLEE [18]. Notice some of the fuzzers, i.e., KLEE, LAF-intel, and AFLGo, can only operate on source code. Thus, we ran them with the source, while we ran the other fuzzers on the compiled binary. For example, we ran AFL in a QEMU mode [33]. To run AFLGo, we gave Line 4 as a target location to give it a guidance.

Figure 1b summarizes the result. All the grey-box fuzzers except LAF-intel failed to find the buggy test case. LAF-intel succeeded because it breaks down the multi-byte comparison statement into multiple single-byte comparisons, which effectively makes code coverage metric sensitive to input mutations. Note, however, LAF-intel was $671 \times$ slower than Eclipser in finding the bug even with source-based instrumentation, which entails lower overhead than binary-level instrumentation.

Notably, the result was even comparable to KLEE. Eclipser was twice slower than KLEE in finding the bug, but Eclipser runs directly on binary code whereas KLEE requires source code. Furthermore, symbolic execution quickly slows down as it encounters more conditional branches because of SMT solving, while complex path conditions do not significantly affect the performance of Eclipser. Indeed, Eclipser achieved even higher code coverage than KLEE on GNU coreutils as we discuss in §V-C, and we also show that Eclipser can scale to handle large real-world applications in §V-E.

This example highlights the potential of grey-box concolic testing. While our technique compromises the precision of white-box fuzzing, it quickly produces test cases for exercising various distinct execution paths of the PUT without relying on any heavy-cost analyses.

III. GREY-BOX CONCOLIC TESTING

Grey-box concolic testing is a way of producing test cases from a given seed input. At a high level, it behaves similarly to dynamic symbolic execution using the generational search strategy [19], [23], where an execution of the PUT with a seed produces a generation of test cases by expanding all feasible branch conditions in the execution path. Grey-box concolic testing operates in a similar manner, but it selectively solves branch conditions encountered in the path while not relying on SMT solving.

The key aspect of our approach is to maintain an independent subset of an approximate path constraint per each input byte of a seed. The constraints help generate distinct test cases that can be used to exercise the same (or similar) execution path of the PUT by resolving the constraints. With such test cases, we can see that some of the conditional branches in the path compare distinct input values even though they take the same execution path. We use such an execution behavior to penetrate conditional branches in a grey-box manner. Our technique effectively resolves branch conditions like white-box fuzzing (i.e., concolic testing), while keeping our system lightweight and scalable like grey-box fuzzing.

A. Overview

Grey-box concolic testing operates with four major functions: SPAWN, IDENTIFY, SELECT, and SEARCH. The crux
Algorithm 1: Grey-box Concolic Testing.

```plaintext
function GreyConc(p, s_p, k)
    pc ← {} // Approximate path constraint
    seeds ← ∅
    execs ← SPAWN(p, s_p, k)
    conds ← IDENTIFY(p, execs)
    for cond in SELECT(conds) do
        s_p, c ← SEARCH(p, k, pc, execs, cond)
        seeds ← seeds + s_p
        pc ← pc ∧ c // Merge two constraints
    return seeds
```

of grey-box concolic testing is expressed in Algorithm 1 with these functions.

**SPAWN** \( (p, s_p, k) \rightarrow \text{execs} 

SPAWN takes in a program \( p \), a seed \( s_p \) and a byte offset \( k \) as input. It first generates a set of \( N_{\text{spawn}} \) distinct inputs by modifying the \( k \)th byte of \( s_p \), where \( N_{\text{spawn}} \) is a user parameter. It then executes \( p \) with the generated inputs, and returns the executions (execs) (see §III-C).

**IDENTIFY** \((p, \text{execs}) \rightarrow \text{conds} \)

IDENTIFY takes in a program \( p \) and a set of executions (execs) as input. It identifies a sequence of conditional statements (conds) that are affected by the \( k \)th input byte (see §III-D).

**SELECT** \((\text{conds}) \rightarrow \text{conds}'\)

SELECT returns a subsequence from the given sequence of conditional statements. In our current implementation of Eclipser, this step simply returns a subsequence of maximum \( N_{\text{solve}} \) randomly selected conditional statements, where \( N_{\text{solve}} \) is a user parameter (see §III-E).

**SEARCH** \((p, k, pc, \text{execs}, \text{cond}) \rightarrow s_p', c\)

SEARCH seeks to penetrate a given conditional statement \( \text{cond} \), and returns a new seed \( s_p' \) that can exercise the new branch at \( \text{cond} \), i.e., the branch not taken by \( \sigma_p(s_p) \), along with a constraint \( c \). The constraint \( c \) represents input conditions to follow the current execution \( \sigma_p(s_p) \). The generated seed takes the same execution up to \( \text{cond} \) as \( \sigma_p(s_p) \), and exercises the opposite branch at \( \text{cond} \) (see §III-F).

At a high level, grey-box concolic testing takes in a program \( p \), a seed input \( s_p \), and a byte position \( k \) as input, and outputs a set of test cases that cover execution paths different than \( \sigma_p(s_p) \). Unlike typical concolic testing, our approach takes in an additional parameter \( k \) to specify which input byte position we are interested in. This is to simplify the process of grey-box concolic testing by focusing only on a single input field located at the offset \( k \). Although our focus is on a single input field, it is still possible to penetrate conditional branches where the condition is affected by multiple input fields, because our strategy may find a satisfying assignment for each one input field at a time. Furthermore, even if \( \text{SEARCH} \) cannot find a satisfying solution, Eclipser performs random mutation to compensate for the error (§IV). Handling such cases in a general fashion is beyond the scope of this paper.

The variable \( pc \) represents an approximate path constraint for the execution \( \sigma_p(s_p) \). Specifically, \( pc \) is a map from a byte in \( s_p \) to an independent constraint for the corresponding byte, which is initially an empty map in Line 2 of Algorithm 1. The approximate path constraint grows as we encounter conditional statements in the execution. Note that this data structure is inspired by independent formulas used in [18], [34].

Grey-box concolic testing instruments every comparison instruction in the execution, but selects only a subset of them in Line 6 for building the constraint \( pc \), thereby, it generates an approximate path constraint. For each of the selected conditional statements, we add the corresponding formula to \( pc \) (Line 9). Note that this process is the same as dynamic symbolic execution except that we maintain an approximated subset of the path constraint.

**B. Example**

To describe our technique, let us revisit the motivating example in §II-C. Figure 2 presents a code snippet taken from the example and the corresponding binary code. We assume that (1) the initial seed file \( s_p \) consists of eight consecutive zeros, (2) \( N_{\text{spawn}} \) is set to 3, and (3) the current offset \( k \) is zero. Eclipser operates by moving around this offset \( k \) throughout a fuzzing campaign as we describe in §IV.

Suppose \( \text{SPAWN} \) generates three inputs \( s_p[0 ← 10], s_p[0 ← 50], \) and \( s_p[0 ← 90], \) and executes \( p \) with the inputs to produce three executions: \( \sigma_p(s_p[0 ← 10]), \sigma_p(s_p[0 ← 50]), \) and \( \sigma_p(s_p[0 ← 90]) \). IDENTIFY then observes from the executions that the first cmp instruction compares the integer 31,337 with three different values in eax: 21, 101, and 181. From the overlapping execution prefix of the three, IDENTIFY returns a pair of the comparison instruction and the following conditional jump instruction. Next, SELECT takes the pair and simply returns it as there is only one item to consider. Finally, \( \text{SEARCH} \) checks the relationship between the three values (10, 50, and 90) and the corresponding compared values (21, 101, and 181) in the overlapping execution. In this case, \( \text{SEARCH} \) infers the following linear relationship: \( eax = 2 × s_p[0] + 1 \). By solving this equation, we obtain 15,668 (0x3d34), which is the value of intInput satisfying the first condition.

However, the solution does not fit in one byte. Thus we have to infer the size of the corresponding input field, which includes the first byte (since \( k = 0 \)) and its neighboring bytes. We consider input sizes up to 8 bytes starting from size 2. In this case, the 2-byte solution works, and it will be used to generate a test case (\( s_p' \)) by replacing the first two bytes of \( s_p \), which results in the following 8-byte file.
in a hexadecimal representation: \( \text{34 3d 00 00 00 00 00 00} \). \text{SEARCH} executes the PUT with this input to see if we can penetrate the conditional branch. Since we can exercise the new branch, it returns the generated seed that contains the approximate path constraint for this branch: \( \{ s_p[0] \mapsto [0x34, 0x3d], s_p[1] \mapsto [0x3d, 0x3d] \} \), where the square brackets represent a closed interval. We describe how we encode an approximate path constraint in §IV-C.

Eclipser now repeats the above processes by using \( s'_p \) as a new seed while incrementing \( k \). When \( k = 4 \), \text{SPAWN} returns the following three executions: \( \sigma_p(s'_p[4 \leftarrow 10]) \), \( \sigma_p(s'_p[4 \leftarrow 50]) \), and \( \sigma_p(s'_p[4 \leftarrow 90]) \). \text{IDENTIFY} finds the correspondence between the fifth input byte (\( k = 4 \)) and \( \text{eax} \).

\text{SEARCH} then figures out that the \text{eax} value monotonically increases with regard to \( s'_p[4] \). It performs binary search by mutating the \( k \)th input byte, and finds out that \( \text{eax} \) changes from -1 to 1, when the input byte changes from \( 0x42 \) (‘B’) to \( 0x43 \) (‘C’). Since we did not find a solution, which makes \( \text{eax} \) be zero, we extend the input field size by one, and perform another binary search between \( 0x4200 \) and \( 0x4300 \). We repeat this process until we find the solution "Bad!", which makes the PUT exercise the true branch of the conditional statement. Finally, \text{SEARCH} produces a seed that contains the string "Bad!".

C. SPAWN

\text{SPAWN} generates test inputs by mutating the \( k \)th byte of the seed \( s_p \) based on the constraint \( s_p[k].\text{constr} \), and returns executions of \( p \) with regard to the generated inputs. The primary goal here is to produce a set of \( N \) test inputs \( \{ i_1, i_2, \ldots, i_N \} \) such that \( \sigma_p(i_1) \approx \sigma_p(i_2) \approx \cdots \approx \sigma_p(i_N) \). Finding such inputs with an SMT solver is feasible in practice, but recall that one of our design goals is to be able to solve approximate path constraints in a lightweight manner.

\text{Eclipser} uses an interval to represent approximate path constraint (see §IV-C). Therefore, finding inputs that satisfy an approximate path constraint is as easy as choosing a value within an interval. If the constraint \( s_p[k].\text{constr} \) was precise as in symbolic execution, then we could always generate distinct test inputs that can be used to exercise the exact same path of the PUT, i.e., we could generate inputs such that \( \sigma_p(i_1) = \sigma_p(i_2) = \cdots = \sigma_p(i_N) \). However, our approach can produce false inputs that do not satisfy the actual path constraint due to the incompleteness of \( s_p[k].\text{constr} \). We note that this is not a serious issue as our focus in \text{IDENTIFY} is on the overlapping execution prefix.

We denote the maximum number of executions to return in \text{SPAWN} by \( N_{\text{spawn}} \), i.e., \( N = N_{\text{spawn}} \). This is a configurable parameter by an analyst. In the current implementation of Eclipser, we set this value to 10 by default, which is chosen based on our empirical study in §V-B. \text{SPAWN} executes the PUT \( N_{\text{spawn}} \) times for a given seed, whereas traditional symbolic execution runs the PUT only once. This is the major trade-off that we have to accept for designing a scalable fuzzer.

D. IDENTIFY

The primary goal of \text{IDENTIFY} is to determine the correspondence between an input byte at the offset \( k \) and conditional statements in \( \sigma_p(s_p) \). It returns a subsequence of \( \sigma_p(s_p) \), which contains all the conditional statements affected by \( s_p[k] \).

To achieve the goal, one may use fine-grained taint analysis. However, it is a memory-hungry process because it assigns an identifier for each input byte, and maintains a set of such IDs for every expression affected by a given input. There are several studies on reducing the space efficiency of fine-grained taint analysis [35], [36], but they assume significant overlaps between set elements. Furthermore, taint analysis instruments every single instruction of the PUT, which can be computationally expensive and too slow for fuzzing.

We use a simple and scalable approach that involves executing the PUT multiple times. Recall that \text{SPAWN} returns \( N_{\text{spawn}} \) executions based on test inputs generated by mutating the \( k \)th byte of \( s \). By observing the behavioral difference in the executions, we can identify the correspondence between the \( k \)th byte and conditional branches in the executions. Specifically, we first extract a set of conditional statements at the same position of the overlapping execution prefixes. We then determine whether a conditional statement \( b \) is affected by the \( k \)th byte of the seed by observing the difference in the decisions of \( b \). This simple approach provides sensitive feedback about which conditional branches in the executions are affected by the input byte.

Note that the imprecision of approximate path constraints is not an issue here, since we can always have executions that partially overlap. Furthermore, since \text{SPAWN} generates inputs by mutation, some of the produced executions may exercise totally distinct execution paths, and thereby, cover interesting paths of the PUT. Eclipser can benefit from such by-products.

Figure 3 illustrates a case where we execute a program \( p \) with two inputs \( i \) and \( i' \) that are different only by the byte value at the offset 10. There are three conditional statements \( b_1, b_2, \) and \( b_3 \) in the overlapping prefixes of the executions \( \sigma_p(i) \) and \( \sigma_p(i') \). In this example, we can observe that the compared values for \( b_1 \) and \( b_2 \) are different in the executions. Therefore, we conclude that the eleventh input byte \( (i[10] \) and \( i'[10]) \) has a correspondence with \( b_1 \) and \( b_3 \).

E. SELECT

During \text{IDENTIFY}, we may end up having too many conditional statements to handle. This phenomenon is often referred
to as a path explosion problem in dynamic symbolic execution. For example, consider the following for loop, where inp indicates a user-supplied input.

```c
for (i=0; i<inp; i++) {
  /* omitted */
}
```

In this case, we can encounter an arbitrary number of conditional statements depending on the user input. If we handle every single statement returned from IDENTIFY, our system may not explore interesting paths for given time.

To cope with this challenge, SELECT randomly selects $N_{solve}$ conditional statements from the given sequence of conditional statements while preserving the order of their appearance. The order should remain the same, because we need to build an approximate path constraint along the program execution. In the current implementation of Eclipser, we use $N_{solve} = 200$, which is determined empirically (§V-B). Note that dynamic symbolic executors such as Sage [19] and KLEE [18] also employ several path selection heuristics to handle the same challenge.

### F. Search

SEARCH resolves a branch condition to cover a new branch in the given conditional statement $cond$. As a result, it returns a new seed as well as a branch condition, which is approximated with an interval (§IV-C), in order for following the current execution path $\sigma_p(s_p)$. The primary challenge here is on solving approximate path constraints without the help of an SMT solver.

Recall that IDENTIFY returns conditional statements that have a relationship with the $k$th input byte. We can represent this relationship as a data flow abstraction, where $s_p[k]$ is an input, and one of operands in each of the conditional statements is an output. The key intuition of SEARCH is that by realizing such an input-output relationship, we can deduce a potential solution of an approximate path constraint.

Specifically, SEARCH focuses on cases where the input-output relationship is either linear or monotonic. This design choice is supported by various previous research works [37]–[39] as well as our own empirical observation. We observed that many conditional branches in real-world programs tend to have a linear or monotonic constraint (see §V-C1).

SEARCH runs in three steps: (1) formulating and solving the current branch condition (§III-F1), (2) recognizing a corresponding input field (§III-F2), and (3) generating a new seed that can penetrate the conditional statement (§III-F3).

1) Solving Branch Condition: Let us assume w.l.o.g. that only one of the two operands of $cond$ is affected by input $i$, and the operand is denoted by $oprnd(i)$. We can decide that the branch condition of $cond$ is linear if there exist $i_1, i_2$, and $i_3$ such that $\frac{oprnd(i_1) - oprnd(i_3)}{i_1 - i_3} = \frac{oprnd(i_2) - oprnd(i_3)}{i_2 - i_3}$. In this case, we can directly construct and solve a linear equation or inequality. On the other hand, $cond$ has a monotonic branch condition if $oprnd$ is a monotonic function over all the observed inputs $i_1, i_2, ..., i_n \ (n \geq 3)$ that executed $cond$. Figure 4 illustrates an example where we have a monotonic input-output relationship between a two-byte input field ($input$) and the compared value ($ebx$). For such a monotonic relationship, we perform a binary search to find out a solution.

2) Recognizing Input Field: Note that our focus so far has been on an input byte, i.e., $s_p[i]$. However, many branch conditions are constrained not only by an input byte but by an input field, e.g., a 32-bit integer or a 64-bit integer. This means SEARCH should be able to handle input fields of arbitrary size. Moreover, our equation solving in SEARCH operates on arbitrary-precision integers, which may give us a solution that does not fit in a byte. We can naturally expand the capability of SEARCH by executing the PUT with several more input candidates. Specifically, we replace the seed with the solution we obtained while considering the solution to have a specific size. When solving linear equations or inequalities, we consider maximum seven cases to try all possible candidates as Figure 5 describes. For binary search on monotonic conditions, we start the search by considering the size of the input field to be one, and then gradually increase the size until a threshold, which is set to 8 in current implementation.

3) Seed Generation: To generate a new seed that executes a new path, we should first approximate the constraint from the current branch, and encode it to the $constr$ field of the newly generated seed. Specifically, we turn the branch condition into a dictionary $c$, which maps an input byte position $i$ to an approximated constraint $c[i]$, which is represented by an interval. For every byte position $i$ in $c$, we update $s_p[i].constr$ with $\neg c[i] \land pc[i]$, where $\land$ represents a conjunction of two intervals. The concrete value of the $s_p[i]$ is also updated with a value that is within the interval $\neg c[i]$. We take the negation of each of the branch condition $\neg c[i]$, because we want to follow the path that is not taken by the current execution. That is, the new seed should take the opposite branch when executed with the PUT. SEARCH returns $c$, and uses it to build up $pc$. We refer to §IV-C for more details on how to approximate the branch conditions found.

### IV. Eclipser Architecture

Although grey-box concolic testing itself enables systematic test case generation for $p$ from a given seed $s_p$ and a byte position $k$, one needs to devise a way to run grey-box concolic testing with varying byte positions as well as with different
Algorithm 2: Main Algorithm of Eclipser.

// function Eclipser(p, seeds, t)
1 function Eclipser(p, seeds, t)
2  $Q$ ← InitQueue(seeds)
3  $T$ ← $\varnothing$
4  while getTime() < t do
5    $R_0, R_1$ ← Schedule()
6    $Q, T$ ← GreyConcolicLoop($p, Q, T, R_0$)
7    $Q, T$ ← RandomFuzzLoop($p, Q, T, R_0$)
8  return $T$

seeds in order to explore interesting paths. This section describes how we tackle such problems in the design of Eclipser.

A. Main Algorithm

Recall from §III-F, grey-box concolic testing currently focuses on linear and monotonic constraints, and it may not be able to handle some complex branch conditions that involve multiple input fields. To cope with these challenges, Eclipser employs a classic grey-box fuzzing strategy. Our goal is to maximize the utility of both grey-box concolic testing and grey-box fuzzing by alternating them. The idea of alternating between fuzzing strategies has been previously proposed [12], [40], [41], and is complementary to ours.

Algorithm 2 describes the overall procedure of Eclipser. Eclipser takes in as input a program $p$, a time limit $t$, and a set of initial seeds $seeds$, and returns a set of test cases $T$ generated during a fuzzing campaign. Eclipser first initializes the priority queue $Q$ with the provided initial seeds $seeds$, and runs in a while loop until the time limit $t$ expires. In Line 5, Schedule allocates resources for grey-box concolic testing ($R_0$) and grey-box fuzzing ($R_1$). Then the two fuzzing strategies, i.e., grey-box concolic testing (GreyConcolicLoop) and grey-box fuzzing (RandomFuzzLoop), alternately generate new test cases until they consume all the allocated resources. We refer to §IV-B for details about the resource management. Eclipser updates $Q$ and $T$ in GreyConcolicLoop and RandomFuzzLoop: it simply adds newly generated test cases, i.e., seeds, to $Q$ and $T$, respectively. $T$ is later returned by the main algorithm when the fuzzing campaign is over (Line 8).

Priority Queue. For each test input generated, Eclipser evaluates its fitness based on the code coverage and adds it to $Q$. Specifically, we give high priority to seeds that cover any new node, and low priority to seeds that cover a new path. We drop seeds that do not improve the code coverage. Eclipser inserts a seed to the queue along with the next value of $k$ to use. Eclipser currently makes $k$ to be both $k - 1$ and $k + 1$, and pushes the seed twice with both positions. One important aspect of the priority queue is that it allows two fuzzing strategies to share their seeds. Note that grey-box concolic testing currently does not extend the size of a given seed when generating new test cases, while grey-box fuzzing can. If the grey-box fuzzing module generates an interesting seed by extending its length, it is shared with the grey-box concolic testing module through the priority queue $Q$.

B. Resource Scheduling

When alternating between the two fuzzing strategies, we need to decide how much resource we should allocate for each strategy. In Eclipser, our resource is the number of allowed program executions. If a strategy runs the PUT more than the allowed number, Eclipser switches the strategy. To decide when to switch, Eclipser evaluates the efficiency of each fuzzing strategy, and allocates time proportionally to the efficiency. Let $N_{exec}$ be the total number of program executions for one iteration of the while loop in Line 4 of Algorithm 2. We define the efficiency $f = N_{path}/N_{exec}$, where $N_{path}$ is the number of unique test cases that executed a new execution path. In other words, Eclipser allocates more resource to the strategy that explores more new paths.

C. Approximate Path Constraint

Recall that grey-box concolic testing approximates path constraints with intervals. An approximate path constraint is a map from an input byte to its corresponding interval constraint: we represent each constraint with a closed interval. Let $[l, u]$ be a constraint $l \leq x \leq u$. Then we can express a logical conjunction of two constraints with an intersection of the two intervals: $[l_1, u_1] \land [l_2, u_2] = [\max(l_1, l_2), \min(u_1, u_2)]$.

Let us assume that SEARCH has resolved a branch condition associated with an $n$-byte input field $x$, and obtained an equality condition $x = k$ as a result. This condition can be expressed with intervals for each byte, without any loss of precision: $\{x_0 \mapsto [k_0, k_0], x_1 \mapsto [k_1, k_1], \ldots, x_{n-1} \mapsto [k_{n-1}, k_{n-1}]\}$, where $k_i = (k \gg (8 \times i)) \& 0xff$ and $x_0, x_{n-1}$ are the least and the most significant byte of $x$, respectively.

Suppose that the resolved branch condition is an inequality condition $l \leq x \leq u$. In this case, the condition is approximated as an interval constraint over the most significant byte of $x$: $\{x_{n-1} \mapsto [l_{n-1}, u_{n-1} + 1]\}$. We only choose the most significant byte here in order to over-approximate the interval represented in “integer” type. Eclipser adds this approximated constraint to $pc$ in Line 9 of Algorithm 1, by performing an element-wise conjunction.

D. Implementation

We implemented the main algorithm of Eclipser in 4.4k lines of F# code, and binary instrumentation logic of Eclipser by adding 800 lines of C code to QEMU (2.3.0) [33]. We wrote the grey-box fuzzing module of Eclipser in F#, which is essentially a simplified version of AFL [4]. We employed the mutation operations used in AFL, and a greedy-set-cover algorithm [14], [42] for minimizing the number of seeds during a fuzzing campaign. To obtain execution feedback from an execution of a binary, we used QEMU user mode emulation because it can easily extend Eclipser to handle various architectures. Currently, Eclipser supports three widely used architectures: x86, x86-64, and ARMv7. Our implementation of Eclipser is publicly available on GitHub: https://github.com/SoftSec-KAIST/Eclipser.²

²The ARMv7 version will not be open-sourced due to an IP issue.
V. EVALUATION

We evaluated Eclipser to answer the following questions:
1) How does the configuration parameter of Eclipser affect its performance? (§V-B)
2) Can grey-box concolic testing be a general test case generation algorithm? If so how does it compare to existing white-box fuzzers? (§V-C)
3) Can Eclipser beat the state-of-the-art grey-box fuzzers? (§V-D)
4) Can Eclipser find new bugs from real-world applications? Is grey-box concolic testing scalable enough to handle such large and complex programs? (§V-E)

A. Experimental Setup

We ran our experiments on a private cluster of 64 VMs. Each VM was equipped with a single Intel Xeon E5-2699 V4 (2.20 GHz) core and 8GB of memory. We performed our experiments on three benchmarks: (1) 95 programs from GNU coreutils 8.27; (2) 4 programs from LAVA-M benchmark; and (3) 22 real-world programs included in Debian 9.1 packages.

First, we selected GNU coreutils to compare Eclipser against KLEE, because KLEE [18] and other white-box fuzzers [25], [27] use this benchmark to evaluate their performance. Second, we evaluated the bug finding ability of Eclipser against grey-box fuzzers on LAVA-M benchmark [43] as it is used to evaluate many existing fuzzers [6], [7], [10]. Finally, we fuzzed real-world applications chosen from Debian 9.1 to measure the practical impact of Eclipser.

Comparison Targets. We chose two existing grey-box fuzzers for comparison, which are available at the time of writing: AFLFast [5] and LAF-intel [28]. We omitted Driller [12] as its current support for ELF binary is limited. We were not able to run VUzzer [7] as it is dependent on IDA pro, which is a commercial product. We also omitted Steelix [6], T-Fuzz [10] and Angora [8] as they are not publicly available.

B. Eclipser Configuration

Recall from §III, Eclipser uses two user-configurable parameters: $N_{spawn}$ and $N_{solve}$. These parameters decide how many branches to identify and to penetrate with grey-box concolic testing, respectively. To estimate the impact of the parameters, we ran Eclipser on each of the programs in the first benchmark (coreutils 8.27) for one hour with varying configurations and measured code coverage differences. In particular, we chose five exponentially increasing values for each parameter.

Figure 6 summarizes the results. When $N_{spawn}$ is too small, IDENTIFY failed to identify some interesting conditional branches, and the coverage decreased as a result, but when $N_{spawn}$ is too large, Eclipser ended up consuming too much time on unnecessary program executions. Similarly, by making $N_{solve}$ too small, Eclipser started to miss some interesting conditional branches, but by making it too large, we started to cover less nodes due to path explosion.

From these results, we decided to use $N_{spawn} = 10$ and $N_{solve} = 200$ as a default set of parameter values for Eclipser, and used them for the rest of our experiments.

![Fig. 6. The impact of $N_{spawn}$ and $N_{solve}$.](image)

C. Comparison against White-box Fuzzing

To evaluate the effectiveness of grey-box concolic testing as a test case generation algorithm, we compared it against KLEE version 1.4.0, which was the latest at the time of writing. We chose coreutils as our benchmark, as it is used in the original paper of KLEE [18]. Out of 107 programs in coreutils 8.27, we excluded 8 programs that can affect the fuzzing process itself, e.g. kill and rm, and 4 programs that raised unhandled exceptions with KLEE. We tested each of the remaining 95 programs for one hour. Additionally, we used the command line options reported in KLEE website [44] to run KLEE. For a fair comparison, we set the same limitation on the input size when running Eclipser. All the numbers reported here are averaged over 8 iterations.

We seek to answer the three questions here: (1) Can grey-box concolic testing itself without the grey-box fuzzing module beat KLEE in terms of code coverage? (2) Can we benefit from alternating between grey-box fuzzing and grey-box concolic testing? and (3) Can Eclipser find realistic bugs in coreutils? How does it compare to KLEE?

1) Grey-box Concolic Testing Effectiveness: We ran Eclipser in two different modes: (1) only with grey-box concolic testing, and (2) only with grey-box fuzzing. The blue and the pink line in Figure 7 present the coverage for each case, respectively. Out of a total 32,223 source lines, grey-box concolic testing covered 20,737 lines (64.36%), and solely using the grey-box fuzzing module covered 18,540 lines (57.54%), while KLEE covered 20,445 lines (63.45%). This result clearly indicates that grey-box concolic testing alone is comparable to KLEE. Note that our tool runs directly on binary executables while KLEE runs on source code. This

![Fig. 7. Line coverage achieved by Eclipser and KLEE over time for coreutils.](image)

2) Comparison against KLEE: In both cases, Eclipser covered more lines than KLEE, as depicted in Figure 7. For grey-box concolic testing alone, Eclipser covered 18,540 lines (57.54%), while KLEE covered 20,445 lines (63.45%).

3) We note that a sharp increase of KLEE’s line coverage around 60 minute does not mean that KLEE starts to rapidly explore code around that point. When a time limit expires, KLEE outputs the test cases remaining in the memory even if their symbolic executions are not finished. Indeed, we further ran KLEE for more than 6 hours, but the coverage increased only by 2.10%.
result empirically justifies our design choice of focusing on solving linear or monotonic branch conditions.

2) Alternation between Two Strategies: The green line in Figure 7 shows the source line coverage achieved by Eclipser while alternating between the two different strategies. It is obvious from the figure that our design choice indeed achieved a synergy: Eclipser covered 23,499 lines (72.93%), outperforming KLEE in terms of code coverage. The standard deviation of Eclipser's coverage was 0.54%, while that of KLEE's coverage was 0.49%. Additionally, Figure 8 shows the coverage difference between Eclipser and KLEE for each program. The x-axis represents tested programs and the y-axis indicates how many additional lines Eclipser covered more than KLEE. The leftmost program is stty, where KLEE covered 66 more lines, and the rightmost program is vdir, where Eclipser covered 554 more lines.

3) Real Bugs from coreutils: The programs in GNU coreutils are heavily tested. Can Eclipser still find some meaningful bugs in them? During the course of our experiments, Eclipser found two previously unknown bugs, each of which can crash b2sum and stty, respectively. On the other hand, KLEE was able to find only one of the bugs during our experiments. This result indeed highlights the practicality of our system.

D. Comparison against Grey-box Fuzzers

How does Eclipser compare to modern grey-box fuzzers? To answer this question, we compared the bug finding ability of Eclipser against state-of-the-art grey-box fuzzers on LAVA-M. Recall from §V-A we were not able to run Steelix and VUzzer for this experiment. Instead, we used the numbers reported in their papers to compare with the other fuzzers. To be fair, we ran the fuzzers with a similar setting that Steelix used. We used the same initial seeds used in [6], and ran our experiment for the same amount of time (5 hours).

Table I shows the number of bugs found from LAVA-M benchmark. The numbers are averaged over 8 repeated experiments. Eclipser found 18.3×, 13.3×, and 4.7× more bugs than LAF-intel, VUzzer, and Steelix, respectively. AFLFast did not find any bug during the experiment. Note that in some programs, Eclipser was even able to find bugs that the authors of LAVA failed to reproduce. For example, in base64, the authors of LAVA could reproduce only 44 bugs in [43]. We note that LAF-intel is a source-based tool, which incurs less instrumentation overhead compared to binary-based tools. For example, when we ran AFL on the LAVA-M benchmark, the number of executions per second with the source-based instrumentation was 9.3× higher than it with the binary-based instrumentation on average. Despite such a disadvantage, Eclipser found far more bugs than LAF-intel. This result shows that grey-box concolic testing can effectively resolve complex conditions to trigger bugs injected by LAVA.

E. Fuzzing in the Real World

We further evaluated our system on a variety of programs in the real world. Specifically, we collected 22 programs from Debian OS with the following steps. First, we used debtags to search for packages containing C programs, which deal with image, audio or video via a command-line interface. Next, we selected the top 30 popular packages based on the Debian popularity contest [45]. We then manually picked only the packages that (1) take in a file as input, (2) can be compiled into an executable, and (3) can be fuzzed with AFLFast without an error. Finally, we extracted at most two programs from each of those packages to obtain a total of 22 programs. We fuzzed each of the programs for 24 hours with a dummy seed composed of 16 consecutive NULL bytes.

Table II shows the results. Overall, Eclipser covered 1.43× (1.44×) and 1.25× (1.25×) more nodes (branches) than AFLFast and LAF-intel, respectively. While investigating the result, we confirmed that grey-box concolic testing of Eclipser indeed played a vital role in achieving high coverage. In oggenc, for instance, Eclipser covered 3.8× more nodes than AFLFast as grey-box concolic testing successfully produced valid signatures for FLAC or RIFF format from scratch.

We further investigated the crashes found, and manually identified 51 unique bugs. In total, Eclipser, AFLFast, and LAF-intel found 40, 10, and 25 unique bugs, respectively. We further analyzed the result, and found that grey-box concolic testing indeed played a critical role in finding bugs. If we ran the same experiment only with the grey-box fuzzing module of Eclipser, which is close to vanilla AFL [4], we obtained only eight unique bugs after 24 hours. This means, grey-box concolic testing helped Eclipser find 5× more unique bugs. We reported all the bugs Eclipser found to the developers, and a total of 8 new CVEs were assigned at the time of writing. We believe this result confirms the practical impact of Eclipser.

VI. DISCUSSION

The current design of grey-box concolic testing focuses on solving branch conditions when the operands of the comparison can be expressed as a linear or monotonic function of an input field. Recall that Eclipser currently resorts to traditional grey-box fuzzing to penetrate branches with complex constraints. This is not a significant drawback since solving
Dynamic analysis is used to complement symbolic execution between the input and the output of a code segment. Such analysis, by simply mutating input data at a source point, can be studied in various contexts. For example, KLEE [18] adopts random path selection, while others [19], [23], [27], [51]–[53] prioritize less traveled execution paths or nodes, or leverage static analyses to guide the search [54]. Although Eclipse follows the similar approach as in [19], we believe adopting more complex strategies is a promising future work. Meanwhile, there are several attempts to increase the scalability of white-box fuzzing, for example by state merging [25], [55], [56]. In contrast, our work mainly focuses on relieving the fundamental overhead for constructing and solving symbolic formulas.

The idea of analyzing programs without expensive data flow analysis has been studied in various contexts. For example, MUTAFLOW [57] detects information flow without taint analysis, by simply mutating input data at a source point and observing if it affects the output data at sink points. Helium [37] uses regression analysis to infer the relationship between the input and the output of a code segment. Such dynamic analysis is used to complement symbolic execution in the presence of unknown library functions or loops. Our work extends these ideas and applies them more aggressively to devise a general test case generation algorithm.

Note that Eclipse currently employs binary-based instrumentation to test a wide variety of programs without source code. However, binary-based instrumentation incurs substantial overhead as we have observed from one of our experiments in §V-D. It is straightforward to improve the performance of Eclipse by adopting source-based instrumentation.

### VII. Related Work

Eclipse is not a fuzzer per se, but it employs a fuzzing module. Therefore, all the great research works on fuzzing [4]–[7], [10], [11], [13], [14], [28], [29], [46]–[50] are indeed complementary to ours.

Since grey-box concolic testing is inspired by white-box fuzzing, it naturally suffers from the path explosion problem. Various search strategies have been proposed to cope with the problem. KLEE [18], for instance, adopts random path selection, while others [19], [23], [27], [51]–[53] prioritize less traveled execution paths or nodes, or leverage static analyses to guide the search [54]. Although Eclipse follows the similar approach as in [19], we believe adopting more complex strategies is a promising future work. Meanwhile, there are several attempts to increase the scalability of white-box fuzzing, for example by state merging [25], [55], [56]. In contrast, our work mainly focuses on relieving the fundamental overhead for constructing and solving symbolic formulas.

The idea of analyzing programs without expensive data flow analysis has been studied in various contexts. For example, MUTAFLOW [57] detects information flow without taint analysis, by simply mutating input data at a source point and observing if it affects the output data at sink points. Helium [37] uses regression analysis to infer the relationship between the input and the output of a code segment. Such dynamic analysis is used to complement symbolic execution in the presence of unknown library functions or loops.

This paper presents a new point in the design space of fuzzing. The proposed technique, grey-box concolic testing, effectively darkens white-box fuzzing without relying on SMT solving while still performing path-based testing. We implemented our technique in a system called Eclipse, and evaluated it on various benchmarks including coreutils, LAVA-M, as well as 22 programs in Debian. We showed our technique is effective compared to the current state-of-the-art tools in terms of both code coverage and the number of bugs found.

### VIII. Conclusion

This paper presents a new point in the design space of fuzzing. The proposed technique, grey-box concolic testing, effectively darkens white-box fuzzing without relying on SMT solving while still performing path-based testing. We implemented our technique in a system called Eclipse, and evaluated it on various benchmarks including coreutils, LAVA-M, as well as 22 programs in Debian. We showed our technique is effective compared to the current state-of-the-art tools in terms of both code coverage and the number of bugs found.

### ACKNOWLEDGEMENT

We thank anonymous reviewers for their feedback. This work was partly supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No.B0717-16-0109, Building a Platform for Automated Reverse Engineering and Vulnerability Detection with Binary Code Analysis), and a grant funded by Samsung Research (Binary Smart Fuzzing).


