ShadowBug
Enhanced Synthetic Fuzzing Benchmark Generation

Zhou, Zhengxiang; Wang, Cong

Published in:
IEEE Open Journal of the Computer Society

Published: 01/01/2024

Document Version:
Final Published version, also known as Publisher's PDF, Publisher's Final version or Version of Record

Published version (DOI):
10.1109/OJCS.2024.3378384

Publication details:
Zhou, Z., & Wang, C. (2024). ShadowBug: Enhanced Synthetic Fuzzing Benchmark Generation. *IEEE Open Journal of the Computer Society*, *5*, 95-106. https://doi.org/10.1109/OJCS.2024.3378384

IEEE Open Journal of the Computer Society

Received 6 March 2024; accepted 13 March 2024. Date of publication 19 March 2024;
date of current version 9 April 2024. The review of this article was arranged by Associate Editor Song Guo.

Digital Object Identifier 10.1109/OJCS.2024.3378384

ShadowBug: Enhanced Synthetic Fuzzing Benchmark Generation

ZHENGXIANG ZHOU AND CONG WANG (Fellow, IEEE)

City University of Hong Kong, Kowloon 999077, Hong Kong

CORRESPONDING AUTHOR: Cong Wang (e-mail: congwang@cityu.edu.hk).

This work was supported by HK RGC under Grant CityU 11217620, Grant 11218521, Grant 11218322, Grant R6021-20F, Grant R1012-21, Grant RFS2122-1S04, Grant C2004-21G, Grant C1029-22G, and Grant N_CityU139/21.

ABSTRACT Fuzzers have proven to be a vital tool in identifying vulnerabilities. As an area of active research, there is a constant drive to improve fuzzers, and it is equally important to improve benchmarks used to evaluate their performance alongside evolving heuristics. Current research has primarily focused on using CVE bugs as benchmarks, with synthetic benchmarks receiving less attention due to concerns about overfitting specific fuzzing heuristics. In this paper, we introduce ShadowBug, a new methodology that generates enhanced synthetic bugs. In contrast to existing synthetic benchmarks, our approach involves well-arranged bugs that fit specific distributions by quantifying the constraint-solving difficulty of each block. We also uncover implicit constraints of real-world bugs that prior research has overlooked and develop an integer-overflow-based transformation from normal constraints to their implicit forms. We construct a synthetic benchmark and evaluate it against five prominent fuzzers. The experiments reveal that 391 out of 466 bugs were detected, which confirms the practicality and effectiveness of our methodology. Additionally, we introduce a finer-grained evaluation metric called "bug difficulty," which sheds more light on their heuristic strengths with regard to constraint-solving and bug exploitation. The results of our study have practical implications for future fuzzer evaluation methods.

INDEX TERMS Bug injection, fuzzing benchmark, symbolic execution, synthetic bug.

I. INTRODUCTION

Fuzzing has emerged as a powerful software testing technique responsible for the discovery of numerous bugs. As a hot research topic, different fuzzing heuristics are being designed to address various scenarios. Correspondingly, fuzzing benchmarks are also being developed to measure the performance of fuzzers. These benchmarks largely fall into two main categories: synthetic and real-world CVE benchmarks. Synthetic benchmarks [1], [2], [3] can contain programs with varying numbers of artificially generated bugs, and the evaluations are commonly based on the metric "number of unique bugs discovered". However, as fuzzing heuristics evolve over time, these benchmarks may become less effective. Specifically, constraint-solving targeted fuzzers, among other advanced fuzzers, may overfit specific implementations of synthetic bugs [4], [5], affecting the accuracy of performance evaluations. Recent fuzzer developments [6], [7], [8], [9] have prompted researchers to shift their focus toward real-world benchmarks containing known CVE bugs.

Although synthetic benchmarks have their drawbacks, we acknowledge the benefits they offer over relying solely on real-world bugs. It is not easy for a large-scale extension of real-world bugs, but synthetic benchmarks can facilitate the creation of bugs without the need for extensive manual labor. Moreover, real-world bugs in different platforms have to be collected repeatedly. On the contrary, synthetic bugs with a single bug generation rule in place, the benchmarks can be readily extended to suit different platforms such as IoT devices or moved onto testing more specific fuzzing heuristics (e.g., to only test the ability to discover buffer overflows by changing the triggering mechanisms of synthetic bugs). We maintain confidence in the potential of synthetic benchmarks and aim to improve the current approaches from two perspectives.

Firstly, there is a shortage of evaluation metrics utilized for synthetic benchmark assessment. The mostly applied metric is the "ground truth", the number of unique bugs, but this metric may lead to indistinguishable experiment results for fuzzer comparisons. Therefore, we propose a new metric called "bug difficulty" that measures the general difficulty to reach and trigger the bug. This metric offers a multi-dimensional evaluation of fuzzer performance. Secondly, we notice that existing synthetic bugs are coarse simulations of real-world bugs (the overfitting problem). They are usually triggered by satisfying a series of comparisons that will benefit fuzzing heuristics specialized in constraint-solving. To address this problem, we recognize implicit constraints, i.e., conditions inferred from program semantics other than comparisons. We propose a transformation from normal inequalities to implicit constraints, enhancing synthetic bugs to resemble real-world bugs more accurately.

To conduct our experiments, we develop a bug-generation prototype named ShadowBug. Our prototype has successfully produced a total of 466 bugs, which we inserted into eight open-source projects to create a comprehensive synthetic benchmark. Using this benchmark, we thoroughly evaluate the efficiency of five renowned fuzzers, which led to the discovery of 391 bugs, thus validating the usability of our approach. Furthermore, we introduce a novel multi-dimensional evaluation metric "bug difficulty" for different fuzzers. This metric assesses the general difficulty of achieving and triggering a bug, highlighting the strengths of various fuzzing heuristics concerning constraint-solving and bug exploitation. Our evaluation metric helps in identifying the most suitable fuzzer for a particular program and facilitates the future study of fuzzer benchmarking.

In summary, we have the following contributions to this paper:

- We improve the fuzzer evaluation with a new metric "bug difficulty" that quantifies the value of a bug, which provides a distinguishable comparison of different fuzzing heuristics.
- We have developed a unique approach that involves transforming regular inequalities into integer overflows that cannot be detected by conventional fuzzers. Through the use of implicit constraints, we are able to create more valuable synthetic bugs.
- We implement the prototype ShadowBug and insert 466 synthetic bugs into eight open-source projects as a new synthetic benchmark. We empirically show that these bugs are robust against advanced fuzzers. In furtherance of our commitment to promoting research in the field, the source code of our prototype tool is available at https://github.com/CongGroup/ShadowBug.

II. MOTIVATION

We have conducted a comprehensive study on existing synthetic benchmarks and observed the following two problems that are crucial to their adoption. In the upcoming sections, we will provide detailed explanations of these problems and elaborate on our proposed solutions.

```
// nresp: the length of input bytes
int nresp = packet_get_int();
if (nresp > 0) {
    // nresp * sizeof(char*) cause
       integer overflow
    char* response = \
        xmalloc(nresp*sizeof(char*));
    for (int i = 0; i < nresp; i++)
        response[i] = packet_get_string
            (NULL);
}
```

FIGURE 1. Bug code from CWE-190 (integer overflow).

A. BUG DIFFICULTY

Problem: Value of bugs: Existing synthetic benchmarks rely on the "ground truth" metric (number of unique bugs), which proves inadequate for conducting comprehensive fuzzer evaluations. For instance, two different fuzzers may identify the same number of unique bugs, yet one may discover bugs that remain undetected by the other. In fact, this problem has commonly emerged in the evaluation results of fuzzers using synthetic benchmarks [10], [11], [12]. It becomes difficult to compare different fuzzers, particularly when their results are virtually indistinguishable. The root cause is that bugs are typically treated equally, regardless of their value or rarity. Yet, rare bugs are more valuable than easily discoverable ones. Although some metrics, such as CVSS [13], are currently employed to assess the value of bugs, they are unsuitable for use as "fuzzing metrics". For example, a serious RCE (remote command execution) bug with a high CVSS rating may be easily discovered by fuzzers due to a loose-fitting condition. As stated in UNIFUZZ [14], we believe that the difficulty involved in detecting a bug is more valuable than the number of bugs identified for fuzzer evaluation.

Solution: Preassigned difficulty: We have created a new metric called "bug difficulty" to measure the value of bugs in our benchmark. This metric includes "exploration difficulty" to determine how hard it is to reach the block containing the bug, as well as "exploitation difficulty" to measure the challenge of triggering the bug. Bug difficulty is calculated based on the execution path latency and the percentage of seeds that satisfy all conditions. Our metric allows for a new set of criteria to compare fuzzers and for justifying the bug distribution in our synthetic benchmark. The bugs in our benchmark are categorized based on exploration difficulty, going from easy-to-reach to hard-to-reach for a smooth transition.

B. IMPLICIT CONSTRAINTS

Problem: Constraint visibility: Another problem with the synthetic benchmark is the constraint visibility. Simply reaching the vulnerable code statement does not always result in triggering the bug. For instance, in Fig. 1, an integer overflow vulnerability occurs when the variable `nrep` $\geq$ 1073741824, and `sizeof(char*)` has a value of 4. This vulnerability

```
void LAVA(int a, int bch, char *s,
          char *d, int n)
{
    int c = a+bch;
    if (a != 0xdeadbeef)
        return;
    for (int i=0; i<n; i++)
        c+=s[i];
    // Out-of-bound write if bch==0
       x6c617661
    memcpy(d+(bch==0x6c617661)*bch,s,n+
        c);
}
```

FIGURE 2. Bug code from LAVA; Bug will be triggered if the comparisons are satisfied.

leads to an out-of-bounds write, which is the chained vulnerability because the size of the buffer `response` is smaller than `nrep`. In other words, the crash will happen only if `nrep` $\geq$ 1073741824; otherwise, the vulnerability will not be triggered. Constraints of this nature are not visible and hidden in vulnerable code statements, such that they can only be inferred via program analysis. We referred to such derived constraints as "implicit constraints". Implicit constraints are common in real-world vulnerabilities. For example, an input byte is the offset of the jump table that controls the called function. The value should be a particular value for the program to call a buggy function. It will be improbable for fuzzers if they can identify these implicit constraints.

However, existing synthetic benchmarks, such as LAVA, only focus on explicit constraints gathered from comparisons, as displayed in Fig. 2. Analyzing the assembly code, the constraints are easily detectable, but the implicit constraints are ignored. Some fuzzing heuristics, particularly those with symbolic execution, enhance code coverage by addressing explicit constraints related to branch-taken. Even though these fuzzers aim for coverage improvements, their heuristics can inadvertently aid in exploiting bugs in existing synthetic benchmarks and result in the overfitting problem. We acknowledge that implicit constraints are necessary for more practical synthetic benchmarks.

Solution: Unobtrusive integer overflow: To address implicit constraints, we drew inspiration from the popular LLVM sanitizer UBSan [15]. UBSan is a compile-time detector that identifies undefined behaviors in a program, including null pointer dereference and signed integer overflow. As these behaviors can often go unnoticed by fuzzers, UBSan usually serves as a complementary tool for fuzzing. We leveraged the invisibility of these behaviors to conceal implicit constraints by activating them once the constraints were satisfied. Among these behaviors, We found that signed integer overflow is particularly suited to this technique.

We have designed a straightforward transformation that converts normal inequalities into integer overflows. For example, if we have an inequality of $a > b$, the integer overflow will be triggered when a exceeds b. This overflow will then be considered the implicit form of the constraint $a > b$ for the generated synthetic bugs. To generate these bugs, we select an input space that can trigger the bug and apply inequalities to slice the search space in the exploitation stage. This helps us to define the corresponding difficulty for the bug. With the separation of explicit and implicit constraints, we can simulate real bugs more accurately.

III. DESIGN

In this section, we will elaborate on the concepts covered in the previous sections and provide clear examples.

A. OVERVIEW

To tackle the shortcomings of current artificial benchmarks, we design and implement the prototype ShadowBug. The fundamental workflow of our approach comprises four stages: *annotation of source code*, *analysis of exploration difficulty*, *generation of synthetic bugs*, and *injection of bugs*. At a high level, ShadowBug interacts directly with the source code of the target programs and inserts synthetic bugs automatically.

Source code annotation: ShadowBug initially performs a static analysis of the targeted program to annotate the beginning and concluding lines of each branch, along with the input functions. These annotations help in aligning the fundamental blocks in the executable files with the branches in the source code. Furthermore, the bug injector depends on these annotations to adjust the source code accordingly.

Exploration difficulty analysis: Once the source code is annotated, ShadowBug proceeds to compile it into an executable file. It carries out a dynamic analysis using the input corpus to obtain the path abstraction [16] and latency of each reachable block. It then utilizes these outcomes to compute the exploration difficulty.

Synthetic bug generation: The process of generating bugs is detached from the target program. ShadowBug manufactures the bug templates in C code via real-world bug patterns, wherein the variables are symbolized. The bug patterns specify the scope of the variables that satisfy the implicit constraints, thus causing the bugs. Depending on the ranges, each bug template's exploitation difficulty is calculated.

Bug Injection: In the final stage, the bug templates will transform into actual bugs by linking symbolized variables to the input bytes that are irrelevant to branch selection. Our bug injector will then use exploration difficulty data to calculate a normal distribution of bug quantity and place the bugs into branches accordingly.

B. SOURCE CODE ANNOTATION

ShadowBug achieves a trickery binary-source matching with the annotations inserted into the source code. In this work, the meaning of code block is considered equivalent to code branch. The code of a dummy program is shown in Fig. 3. At the start of each branch in the source code, we assign a random branch ID to a shared variable `BrID` which is unique for each branch. We then store the corresponding line number of each `BrID` value to quickly pinpoint a branch in the source code.

```
int main () {
    char buffer[30];
    int *BrID=shmat(shmid, NULL, 0));
    FILE *pFile = fopen("input","rb");
    ...
    a = buffer[3] + 15;
    b = buffer[4] - 7;
    *BrID = 0x76124;
    if(a*b + 2*a + b < 200){
        *BrID = 0x706e67;
        if(!strncmp(buffer,"png",3)){
            *BrID = 0x55;
            if (buffer[5] > '7'){
                // A crash
                abort();
    }}}
    return 0;
}
```

FIGURE 3. Dummy program containing a magic-byte check and a comparison.

When running the annotated binary with dynamic instrumentation, the `BrID` is updated when entering a new branch. The instrumentation code accesses this shared variable and records the matching of the `BrID` and the block address (e.g., pair (*0x706e67*,*0x4009d0*) for the `strncmp` branch). It then stores the matching results of the line number of the source code, `BrID`, and the block addresses in a matching table for subsequent phases. Additionally, ShadowBug also annotates input functions to assist in inferring branch-taken-related input bytes.

C. EXPLORATION DIFFICULTY ANALYSIS

Definition: According to some studies on fuzzing [4], [17], the fuzzing process can be divided into two stages: exploration and exploitation. During the exploration stage, the fuzzer attempts to reach the vulnerable block by satisfying the branch-taken conditions; in the exploitation stage, it conducts constrained fuzzing that mutates the input to trigger the crash while also preserving the reachability of the target block. For a fuzzer, the ability to explore the path or exploit a bug largely depends on two factors: execution time per fuzzing round and the size of the search space. Many fuzzer designs aim to improve these two properties [11], [17], [18], [19], [20], [21], [22], [23], [24]. Therefore, we have defined the exploration difficulty of a branch as:

$$Df(Br) = (1 + latency) * log\frac{Sp(\emptyset)}{Sp(Br)}$$

where `latency` is the normalized number of instructions executed along the path to this branch, `Sp(Br)` is the input space that satisfies all constraints from the entry point to the branch. `Sp(∅)` refers to the search space consisting of all possible inputs.

The rationale behind the definition of exploration difficulty is simple: the more latency it takes to reach a branch, the more time the fuzzer will need to spend on trial-and-error, and the more difficult that branch is considered to be. We get the latency value by conducting a min-max normalization on the number of executed instructions. Another component in the definition is the ratio between the overall search space and the feasible input space that satisfies the constraints before the branch. This ratio reflects the anticipated number of trials required to bypass all preconditions along the path. Intuitively, the exploration difficulty increases as the anticipated number of trials needed to bypass preconditions grows.

Algorithm 1: Computing Exploration Difficulty.

Input: Ic: input corpus
Output: Df: matchings of BrID and difficulty

1: $Df \leftarrow \emptyset$
2: $brsp \leftarrow \emptyset$
3: **for** each $seed \in Ic$ **do**
4: $BrID\_path \leftarrow dynInst(seed)$
5: **for** each $(Blk, BrID, latency) \in BrID\_path$ **do**
6: $pc, tinbytes \leftarrow collect\_constraints(Blk)$
7: $brpc \leftarrow path\_abstract(pc, tinbytes)$
8: **for** each $byte \in tinbytes$ **do**
9: $min, max \leftarrow SMT\_opt(byte, brpc)$
10: $brsp \leftarrow brsp \cup min \leq byte \leq max$
11: **end for**
12: $wsp \leftarrow 256^{\|tinbytes\|}$
13: $Diff \leftarrow (1 + latency) * log(\|wsp\|/\|brsp\|)$
14: $Df \leftarrow Df \cup (Diff, BrID)$
15: **end for**
16: **end for**

Path abstraction: ShadowBug leverages "path abstraction" for exploration difficulty computation. Path abstraction collects constraints along an execution path and summarizes the feasible search space. In this work, we extend the path abstraction in an advanced fuzzer Pangolin [20], which has optimized the objective expressions of a state-of-the-art abstraction inference method BWAI [16] for fuzzing. BWAI formulates the abstraction as an SMT-based optimization problem (SMT-Opt) [25]. It enumerates all linear relationships between input variables and solves these linear expressions subject to path constraints to get the boundaries of each input byte. Moreover, the enhanced method reduces the objective expressions to only the input variables and the linear part of each path constraint.

Computing exploration difficulty: The algorithm for computing exploration difficulty is illustrated in Algorithm 1 and comprises three steps. *Step 1: collecting the information.* At line 4, the target program will be executed with the input corpus and dynamic instrumentations where some blocks will be traversed. The instrumentations will generate a "`BrID` path" recording the order of visited blocks, their corresponding block addresses and branch IDs, and the average latency to reach the branches for the first time.

At line 6, ShadowBug statically collects the constraints for each branch in the path, which we call "pre-constraints". Meanwhile, it also conducts a static taint

analysis on the input variables to filter out the input bytes involved in the constraints. Take the path to the crash in the dummy program in Fig. 3 as an example. The path contains three constraints (`a*b + 2*a + b < 200`, `!strncmp(buffer, "png", 3)`, and `buffer [5]>'7'`) with latencies of 0.26, 0.21, and 0.45 respectively (min-max normalization on instruction counts will be performed after all reachable blocks have been visited).

Step 2: solving constraints to get the solution space size. At line 7, using the pre-constraints for a branch, ShadowBug applies path abstraction to the tainted input bytes to get the post-constraints. Path abstraction involves multiple SMT-Opt problems with the objective expressions comprising the tainted input bytes and the linear sections of each pre-constraint. The SMT solver then generates boundaries for the objective expressions, which we call "post-constraints". For the BrID path in the dummy program, the post-constraints will be `buffer [0-2] == "png"`, `2*buffer [3]+buffer [4] ≤ 178`, and `buffer [5] ≥ 0x37`. We can compute the size of the solution space by calculating the number of feasible values within the post-constraints. However, counting the feasible values directly proves challenging due to the high dimensionality of the solution space, - linear constraints like `2*buffer [3]+buffer [4] ≤ 178` can have more than one variable.

To enhance speed, at lines 8–10, a super-set of the genuine solution space is acquired with a trade-off in precision. By making the tainted input bytes as new objective expressions and solving them subject to the post-constraints, we derive fresh boundaries for each input byte, resulting in a super-set of the initial solution space. Usually, the post-constraints with more than one variable will be changed. For example, we get the final boundaries of the two bytes buffer [3], and buffer [4] by solving the SMT-Opt problem with respect to the only related constraint `2*buffer [3]+buffer [4] ≤ 178`. The results will be `0 ≤ buffer [3] ≤ 89` and `0 ≤ buffer [4] ≤ 178`. The corresponding solution space size will be 90 for buffer [3] and 179 for buffer [4]. Although this approach avoids solving too complex constraints, it comes at the cost of lower accuracy in exploration.

Step 3: calculating the exploration difficulty. As shown in line 12, since each byte can only have 256 feasible values, the total number of possible inputs is 256 to the power of the number of tainted input bytes. The size of the solution space is equal to the product of the count of feasible values within the boundaries for each input byte. For the branches in the aforementioned BrID path, the solution space size will be $90 \times 179 = 16110$ for `a*b + 2*a + b < 200`, 1 for `!strncmp(buffer, "png", 3)`, and 72 for `buffer [5]>'7'`. In line 13, we estimate the expected number of fuzzing trials needed to bypass all constraints by calculating the ratio of the solution space size to the search space size for the tainted input bytes, which are $2^{16}/16610 = 3.95$, 2^{24}, and $2^8/72 = 3.56$ for the branches in the BrID path separately. Considering the multiplication in this ratio can be potentially large, we take the logarithm and multiply it by $(1 + latency)$ to compute the difficulty of one branch, and the exploration difficulties are $1.26log(3.95)$, $1.21log(2^{24})$, and $1.45log(3.56)$. Finally, we add the results and their corresponding branch IDs as pairs to the difficulty set for use in subsequent stages.

TABLE 1. Transformations From Normal Constraints to Integer Overflows. We Only Check the Overflow Flags of Operations Highlighted in Red

P/N	normal	signed overflows
positive	exp>C	exp+(2^{B-1} - C)
	exp<C	exp-C -2^{B-1}
negative	exp>C	exp-C + 2^{B-1}
	exp<C	exp-(2^{B-1}+C)

D. SYNTHETIC BUG GENERATION

Implicit constraints: ShadowBug transforms implicit constraints into integer overflows. These constraints possess the trait of being identical to ordinary code syntax. Integer overflows are widely applied in assembly code and they are not always considered flaws, thus making them nearly undetectable. For instance, the `jo` and `jc` conditional jump instructions rely on the overflow flags for normal control flow branching. Moreover, certain compilers may purposely include overflows in comparisons for optimized performance. As a consequence, fuzzers cannot simply recognize implicit constraints by identifying integer overflows in the programs.

To transform a constraint of the form `exp > C` into an integer overflow, the constraint becomes `(`2^{B-1}`-C) + exp`. Here, `exp` is an expression with respect to input variables, and the variable `B` is the bit length of the data type in the constraint being used (such as 8 for *char*, 16 for *short*, and 32 for *int*). The constant `C` is a positive signed value of `B` bits. If `exp` exceeds the value of `C`, the expression `(`2^{B-1}`-C) + exp` will be larger than 2^{B-1}, which will set the signed overflow flag in the CPU status register. By examining the status of the overflow flag after the addition, we can determine whether the variable `exp` satisfies the constraint. For negative values of `C` and the inverted inequality `exp < C`, similar transformations take place. Table 1 provides a summary of all potential expression transformations.

Bug patterns: With the implicit constraint transformations, we can generate synthetic bugs using complicated real-world bugs as the reference. When it comes to fuzzing, solving both explicit and implicit constraints is crucial. Therefore, we have created a constraint-only bug pattern to simulate real-world bugs. The pattern ignores semantics in the reference programs and only keeps implicit constraints related to the crash triggering. For example, the integer overflow bug in Fig. 1 has the implicit constraint `nresp` > $\frac{2^{32}-1}{4}$. If we only preserve the bit length, we can create the pattern that `input` > $\frac{2^B-1}{B/8}$. This pattern summarizes that the root cause of this common

TABLE 2. Bug Patterns From Common Vulnerabilities. $\text{Input}_{B/8}$: Input of *B/8* Bytes; B: Number of Bits; R: Random Value

Description	Sample Code	Bug Pattern
Particular matching	strncmp(input, "png", 3)	input == rand
Wrong size calculation	malloc((int)n*sizeof(int))	$\text{input}_{B/8} > \frac{2^{B-1}}{B/8}$ (signed)
Extreme value	char a = b+3; # b >124	$\text{input}_{B/8} > 2^{B-1}$-R-1
Wrong type casting	unsigned char b; char a = (char)b; # b >127	$\text{input}_{B/8} > 2^{B-1}$+R

integer overflow bug is the computation of the buffer size through a multiplication of the number of variables and the length of the data type. Our bug pattern is essentially the experience learned from common bugs with the complicated context discarded, and this requires manual efforts to study different patterns.

A bug pattern will specify the symbolized input variables, alternative constants, and some random noises for differentiated results. The symbolized input variable needs to declare the number of bytes occupied. In the aforementioned example, `nresp` will be learned as a four-byte input variable. Additionally, the variable `B` is the alternative constant that can be 16 or 32. We study some popular bugs and the CWE(Common Weakness Enumeration) and summarize in total four bug patterns suitable for our prototype as shown in Table 2. The patterns summarize common integer overflows, type casting from unsigned values to signed values, and extreme values that easily cause overflows. The sample code for wrong size calculation is a *malloc* function similar to that in Fig. 1. For the extreme value and wrong type casting, the sample code shows how integer overflow may happen if the implicit constraints are satisfied. We generate synthetic bugs by combining the bug patterns. The pattern may be repeated in one bug, but normally a bug will have no more than two patterns so that the difficulty to exploit the bug will be affordable.

E. BUG INJECTION

Concretize implicit constraints: A synthetic bug is generated by concretizing the bug patterns as code that can trigger crashes in the target program. ShadowBug will randomly select some untainted input bytes that are irrelevant branch-taking. The input variables directly read the values from these bytes and assemble the bytes into larger data types such as *short* and *int*. To avoid overlapped input bytes that cause un-triggerable bugs, we intentionally keep the input bytes used in one bug pattern independent of those in others. With the vulnerable code inserted into the target program, fuzzers can trigger the crash by mutating the corresponding input bytes used for concretization. The alternative constants in the bug patterns will be randomly selected from the choices, and the noise will be a small integer that will not affect the experience summarized in the bug pattern. For ease of understanding, we will show the bug concretization of a composite bug pattern ($\text{input}_{B/8}$ > 2^B-1-R) && ($\text{input}_{B/8}$ < 2^B). We set the alternative constant to be 16, the random noise to be 3, and choose the first and fourth bytes input bytes in the input file. We will have the code for concretized implicit constraints: `((in[0]+(in[3]<<8)) > 65533) && ((in[0]+(in[3]<<8)) < 65536)`.

Difficulty of a bug: The difficulty of a bug is defined as the sum of exploration and exploitation difficulty. Similar to exploration difficulty, exploitation difficulty should also consider latency and the ratio between the solution space and the whole search space. However, synthetic bugs generated from bug patterns are often short and take no more than ten milliseconds to run. For ease of use, we omit latency and only calculate the search space ratio as exploitation difficulty - $ExplotDf(Br) = log\frac{Sp(InB)}{Sp(Bug)}$. This can be done in advance when bug patterns are concretized.

Bug insertion: Ideally, we expect that the number of bugs and the exploration difficulties fit the normal distribution. ShadowBug sorts the exploration difficulties of reachable branches and grades them into three levels (low, medium, and high) based on the tertiles. We then calculate the mean and standard deviation for the normal distribution of the sorted difficulties, and the ratios of the three under-curve areas of the distribution within the intervals `[0, 1st tertile]`, `[1st tertile, 2nd tertile]`, and `[2nd tertile, ∞]`. According to the area ratios, we insert a predefined number of bugs (randomly between 1–3 bugs) to the blocks of low difficulty (lowest tertile), and the corresponding number of bugs to the medium and high difficulty blocks. In this way, we guarantee the generated bugs are well-arranged with respect to different levels of exploration difficulty. It is noteworthy that the bug code is inserted at the last line of each block in case the bugs in the ancestor node are triggered and those in successor nodes will be stashed due to the early quit of the program. Regarding exploitation difficulties of the bugs, we equally insert bugs of easy and hard implicit constraints to each exploration difficulty level. In addition to visited blocks, there are also unreachable blocks that our input corpus cannot cover. We insert some synthetic bugs into these blocks as surprises, although we cannot precisely measure the difficulties of these bugs.

PoC generation: Unlike existing synthetic benchmarks, ShadowBug specifies input bytes and implicit constraints to trigger synthetic bugs, allowing us to generate corresponding Proof of Concept (PoC) input seeds with symbolic execution.

Basically, we generate a "PoC seed" that satisfies the implicit constraints in a concretized bug pattern. Then we find the seed from the input corpus that traverses the corresponding branch containing the bug. Considering that only untainted input bytes are used for implicit constraints, we directly overwrite concretization bytes in the input seed with corresponding values from the PoC seed. PoC generation can be automated by recording input seeds and the blocks they traverse.

IV. IMPLEMENTATION

This section covers the implementation-specific details of ShadowBug. The prototype integrates several third-party tools and an automatic toolchain in Python scripts.

A. BINARY ANALYSIS

We parse the source code by `Clang Libtooling` [26] which provides full control of the Clang AST tree. We enumerate `if`, `for-loop`, `while`, and `switch` statements as candidate branches, and insert statements that assign random values to the shared variable *BrID* as the first line of each branch. We also statically search function calls to input functions (such as `fread`, `fopen`, and `read`) and extract names and sizes of input buffers. If input functions are unrecognizable, an interface is provided for manually specifying input buffer information. We store input variable information in a matching table that records exploration difficulty and branch ID pairs.

We use the dynamic instrumentation tool `Intel-Pin` [27] to match block addresses with branch IDs and record the average latency of each visited block. When the execution trace enters a block, the instrumentation accesses shared variable `BrID` and matches the value with the block address. Latency is recorded by a count function inserted before each instruction. Note that a branch may be visited by different input seeds and the latency can also vary due to different activated program functionalities. We handle this by simply taking the average of all latencies belonging to one block. However, we can achieve more fine-grained difficulty by storing input-specific latency and using it for future promotion.

We conduct static taint analysis and path abstraction on each seed in the input corpus using Angr [28], which supports symbolic execution and static binary analysis. We utilize taint analysis to filter out input bytes that are essential for branch-taking but only for the first 200 bytes of an input file, as input headers are commonly parsed for branch-taking. Although taint tracking is sharable among the successors of a branch, it can take up a lot of time to track all input bytes, especially in the case of large binary files such as the `readelf` input file, which can be several megabytes in size. On the other hand, explicit constraints are collected by Angr for the path abstraction, and we execute the symbolic execution at the granularity of each input byte. The linear parts of all path constraints are extracted through a BFS on the AST tree. Additionally, we compute the power of each expression in path constraints and the number of influenced input bytes of each term. We remove non-linear parts, which have a power larger than 2 or multiplication of more than one input byte, by multiplying them with zero. Furthermore, the SMT solver of Angr is used to calculate pre-constraints and post-constraints of tainted input bytes, and exploration difficulties are also computed.

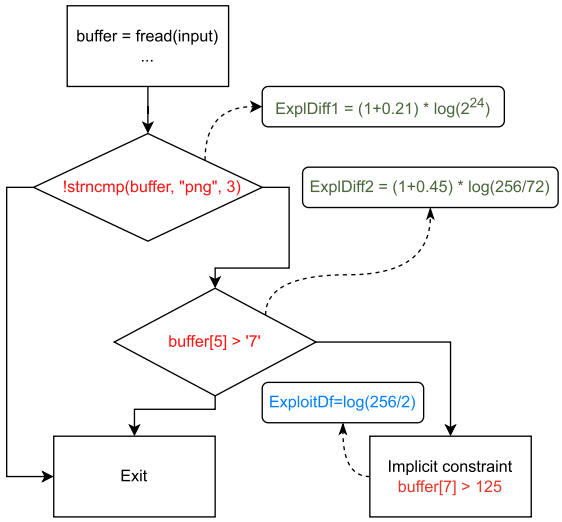


FIGURE 4. Bug difficulty of one synthetic bug in the dummy program. The synthetic bug has an implicit constraint that buffer [7] > 125 and it will crash the program if it is satisfied.

```
#define is_OF(x) ((x&0x0800)>>11)
...
char x = buffer[7];

// Implicit form of constraint x >= 100
// Integer overflow
long GT_0_flag;
unsigned char GT_0;
asm( "push %%rax\n\t": );
asm( "add $28,%%al\n\t"
      :"=a" (GT_0)
      :"a"(x));
asm( "pop %%rax\n\t": );

// Access the status register
asm( "pushf\n\t"
    "pop %%rax\n\t"
      :"=a" (GT_0_flag));

// GT_0 is 0 if overflow does not
   happen
GT_0 *= is_OF(GT_0_flag);

// Bypass implicit constraint and crash
memcpy(buffer+0x100000*GT_0, buffer, 1)
    ;
```

FIGURE 5. Implementation of implicit constraints based on integer overflows.

B. SYNTHETIC BUG

We use inline assembly code to implement integer overflows in implicit constraints, which enables direct access to the status register through the `pushf` and `pop %%reg` instructions. Fig. 5 provides the code for the implicit constraint `x` $\geq$ `100`, where the variable `x` reads a value from `buffer [7]` and is subsequently added by 28. This code is equivalent to the code

```
char x = buffer[7];
// Buffer overflow if x>=100
memcpy(buffer+0x100000*(x>=100), buffer
    , 1);
```

FIGURE 6. Explicit bug equivalent to the above implicit constraint that a buffer overflow will be triggered if x>=100.

in Fig. 6. If x is greater than or equal to 100, the addition at line 10 will cause a buffer overflow. The "sign variable" GT_0 stores the result of the addition, which can be used in another implicit constraint as the input variable. We leave the construction of chained constraints as potential future work. At line 16, the overflow flag is checked by accessing the status register. If the overflow does not occur, GT_0 will be set to zero at line 21. Finally, the bug is triggered in the same way as other synthetic benchmarks, with the code at line 24 accessing invalid memory if GT_0 is not equal to 0. In general, we successfully trigger the bug at line 24 only if the constraint $x \geq 100$ is met. Furthermore, if a synthetic bug has more than one implicit constraint, e.g., another sign variable GT_1, we will modify the vulnerable function call at line 24 to `memcpy(buffer+0x100000*GT_0*GT_1, buffer, 1)`. The bug can be triggered only if both constraints are satisfied.

V. EVALUATION

In this section, we will present the experimental results of the synthetic benchmarks generated by ShadowBug. Our evaluation focuses on three key aspects:

- Our first focus is on the effectiveness of implicit constraints to make sure they indeed introduce challenges to fuzzers for particular conditions. To evaluate this, we transformed the last two constraints of the dummy program (i.e., `if (!strncmp(buffer, "png", 3))` and `if (buffer [5] > '7'))` to their implicit forms and evaluated them against five different fuzzers. We also evaluated these fuzzers with the original dummy program to compare the results. The outcomes reveal that implicit constraints are notably more challenging to satisfy than explicit constraints and bugs can only be triggered if both explicit and implicit are satisfied.
- Our next assessment is on the robustness of synthetic bugs generated by ShadowBug. We report the discovery of unique bugs regarding exploration and exploitation difficulty. The visualization of bug difficulties highlights the distinguishable features of various fuzzing heuristics. Additionally, we justify the robustness of the synthetic benchmark using only a portion of the bugs discovered, as the most difficult bugs remain undiscovered.
- Finally, to support the large-scale extension of ShadowBug to other projects, we evaluate the average overhead caused by the additional bugs introduced into the target programs. Our results indicate that the overall overhead is reasonable, even in the presence of dozens or hundreds of bugs in a single program.

TABLE 3. Time in Seconds for the Fuzzers to Discover the Dummy Bug Against the Explicit Constraints and the Implicit Forms of the Last Two Constraints in the Dummy Program. "inf" Indicates the Fuzzing Campaigns Cannot Find the Bug Within the Timeout

Type	AFL	AFLFast	AFL++	Honggfuzz	QSYM
Explict	2130.7	2382.0	471.3	1522.0	154.3
Implicit	Inf	Inf	Inf	Inf	Inf

Expriment setup: The benchmark consists of eight programs from Binutils and Google fuzzer test suite with in total of 466 recognized bugs (along with some uncounted "surprise" bugs in the target programs). We collect the input corpus from different open-sourced repositories to ensure the basic coverage for difficulty calculation. The benchmark is evaluated against five eminent fuzzers: AFL [23], AFLFast [29], AFL++ [7], Honggfuzz [30], and QSYM [11]. Each fuzzing campaign will be repeated five times within 72 hours. All experiments were conducted on a Ubuntu 18.04.6 LTS server with an 18-core Intel(R) Xeon(R) W-2295 CPU @ 3.00 GHz and 128 GB RAM. We allocated two CPU cores for the QSYM fuzzing campaign, with one dedicated to the fuzzing engine and another for the SMT solver. The remaining fuzzers were run in parallel mode with two CPU cores to ensure fairness. The input seed is randomly picked from the input corpus and remains constant across different fuzzing campaigns for the same program. For the experiments on the dummy program, the fuzzing begins with an empty file.

A. EXPLICIT VS. IMPLICIT

Table 3 displays the time required to discover the bug while fuzzing the dummy programs. The data in two rows indicate the average duration for explicit and implicit constraints. We observe that explicit constraints take considerably longer, with AFL, AFLFast, and Honggfuzz taking several dozen minutes to bypass the magic-byte check. These fuzzers do not possess any heuristics to solve explicit constraints, resulting in the need to enumerate all possibilities as each fuzzing campaign begins from an empty file. In comparison, AFL++ integrates LAF-Intel [31] and REDQUEEN [10] to bypass hard constraints. QSYM, on the other hand, can effortlessly generate the seed to bypass explicit constraints through symbolic execution.

However, the results in the table indicate that none of the fuzzers could discover the bug protected by implicit constraints (denoted as "Inf" in the table). This is largely attributed to the fact that these five fuzzers (and most state-of-the-art fuzzers) rely on coverage-based seed scheduling while implicit constraints remove branches from the original control flow. From the perspective of the fuzzers, fuzzing the implicit constraints is equivalent to black-box fuzzing. To bypass implicit constraints, the fuzzers would need to search for a small portion of input seeds from the 2^{40} possibilities, which is usually impossible. Our experimental findings suggest that implicit constraints are "invisible" to existing fuzzers, regardless of their heuristics.

TABLE 4. Number of Bugs Found by Each Fuzzer With Respect to Each Target Program

# Bugs	AFL	AFLFast	AFL++	Honggfuzz	Qsym
readelf	16	19	22	19	20
objdump	16	16	25	19	23
tiff2pdf	22	23	37	27	35
xmllint	75	72	104	80	106
sqllite3	80	79	106	83	97
nm	15	17	28	19	25
libpng	22	21	28	24	24
json	11	13	23	18	19
Total	257	260	373	289	349

B. EVALUATIONS REGARDING BUG DIFFICULTY

The target programs underwent an insertion of 466 triggerable bugs, which were verified through automatically generated PoCs. It was observed that 391 (83.9%) of these bugs were successfully discovered by at least one fuzzer. Table 4 displays the number of synthetic bugs found by fuzzers against the target programs. From the results, it is evident that AFL++ and QSYM discovered a noticeably larger number of bugs (about 100 bugs) compared to the others. Considering the bug quantity standards, AFL++ is considered the best among all fuzzers since it managed to locate the most bugs (373 out of 466). Specifically, AFL++ discovered 37 unique bugs which were not found by other fuzzers; whereas, QSYM discovered 15 unique bugs, and Honggfuzz only found three. AFL and AFLFast, on the other hand, were unable to discover unique bugs.

Instead of relying solely on bug quantity as the metric, a multidimensional comparison of bug difficulty offers a more comprehensive visualization of fuzzing performance. Fig. 8 shows the heatmaps that demonstrate the number of bugs discovered by the fuzzers concerning both exploration and exploitation difficulty. Since the bug difficulty is calculated based on the number of executed instructions, it cannot be compared among different binaries. We divide the difficulty of each binary into three levels based on the tertile of the values of exploration and exploitation difficulty. The three levels of bug difficulty from low to high are low(L), medium(M), and high(H). The deeper the color is in the heatmap, the more bugs are found in the blocks of the corresponding bug difficulty. The heatmaps indicate that AFL++ has deeper colors in the right-bottom part (59 bugs at the H-L grid), denoting that it is more adept at exploiting bugs. On the other hand, QSYM has a deeper color in the top part of the heatmap(36 bugs at the L-H grid), confirming its excellent fuzzing heuristic in exploring new branches.

In addition, we can also only focus on either exploration or exploitation difficulty. Fig. 7 illustrates that the number of bugs found by each fuzzer decreases as the exploration and exploitation difficulty increases. We notice that QSYM and AFL++ find the most bugs at the three bug difficulty levels.

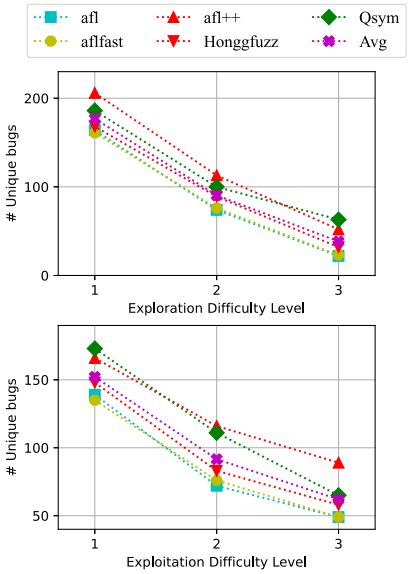


FIGURE 7. Bugs found with respect to exploration and exploitation difficulty. Bug difficulties have three levels based on the tertile of bug difficulty for each binary.

Moreover, as QSYM is a hybrid fuzzer targeting coverage improvement, it outperforms AFL++ when the exploration difficulty is at the highest level (level 3). On the other hand, AFL++ discovers far more bugs (36.9%) of the highest exploitation difficulty than other fuzzers, indicating that it may have a more powerful mutation strategy after reaching the vulnerable block.

C. OVERHEAD

Table 5 showcases the average overhead of synthetic bugs in the eight target programs. Maintaining a low overhead is critical to support a large-scale extension of synthetic bug generation. The static analysis process requires 7.5 seconds for a single bug, encompassing both difficulty calculation and the generation of bugs from templates. It is discernible that the predominant time-consuming factor lies in path abstraction, particularly when symbolic execution is invoked. Given that static analysis is performed only once for each target program, the incurred time cost is deemed acceptable. From the experimental results, each bug, on average, adds 0.04% latency to the program and takes up only 9 KB of space. The current overhead is empirically affordable. The low overhead can be attributed to two primary reasons. Firstly, the code template for each implicit constraint, which consists of a few assembly instructions, remains fixed regardless of the program context. Hence, the synthetic bug's overhead is mainly determined by the number of implicit constraints, and its effect can be managed by modifying the exploitation difficulties. Secondly, the bugs are inserted into different paths from the input corpus, and although several implicit constraints are added to the programs, only a few will activate within a single execution trace. This further ensures the potential scalability of ShadowBug for large-scale extensions.

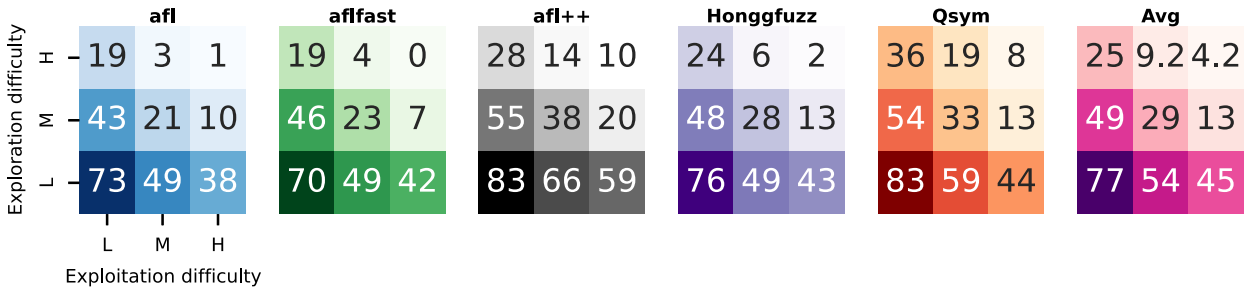


FIGURE 8. Bugs found with respect to exploration and exploitation difficulty. Y-axis: The exploration difficulty from Low to High; X-axis: The exploitation difficulty from low to high. The color depicts the fuzzer's ability, whereas deeper colors indicate greater efficacy in fuzzing bugs of the corresponding difficulty.

TABLE 5. Overhead of Each Target Program. CPU Considers the Extra Latency Brought by the Inserted Bugs, While Storage is About the Additional Disk Overhead. SA Stands for Static Analysis

	readelf	objdump	tiff2pdf	xmllint	sqllite3	nm	libpng	json	average
#bugs	32	42	58	146	138	41	48	32	
CPU	+1.2%	+0.3%	+2.4%	+0.9%	+0.7%	+3.5%	+4.1%	+2.9%	+0.04% / bug
Storage(MB)	0.3	0.4	0.5	1.3	1.2	0.3	0.4	0.3	9.0K / bug
SA(s)	384	896	441	362	574	227	393	216	7.5s / bug

```
// constructing three variables GT_0,
   GT_1, GT_2 in the same way as the
   example
...
// UAF: Satisfy GT_0 to free a buffer
PTR = {normalPointer, pointerToBuffer}
free(PTR[GT_0]);
// Satisfy GT_1 to reuse the buffer
use(PTR[GT_1]);

//Null-pointer dereference
PTR = {normalPointer, nullPointer};
memcpy(PTR[GT_2], buffer, 1);
```

FIGURE 9. Examples of UAF and null-pointer dereference.

VI. DISCUSSION

A. DIVERSE BUG TYPES

ShadowBug achieves the invisibility of bugs based on integer overflow. There are concerns about the potential overreliance on integer overflow, a situation that could lead to a dearth of bug-type diversification. It is noteworthy that integer overflows constitute merely a fraction of the implicit constraints, but bug type is contingent upon the specific triggering mechanism employed. Illustrated in Fig. 5 is an exemplification of an implicit bug, where the final statement `memcpy(buffer+0x100000*GT_0, buffer, 1)` dictates the bug type as a buffer overflow. The preceding statements serve to generate the implicit conditions requisite for triggering the bug. This example can be easily manipulated to manifest bugs of varied types through minor adjustments in the triggering mechanism. Fig. 9 elucidates two instances of this adaptability, showcasing use-after-free (UAF) and null-pointer dereference scenarios. Once the implicit constraints of GT_0 and GT_1 are satisfied, PTR[1] will be freed, after which it will be used to trigger the UAF. Similarly, the null-pointer dereference will be triggered upon the satisfaction of GT_2 constraints. The design of ShadowBug prioritizes high flexibility, enabling seamless extension with copious semantic information and diverse bug-triggering mechanisms.

B. SEMANTIC LOSS

ShadowBug utilizes constraint-only bug patterns that neglect the program's semantics and concentrate on the abstract bug-finding level. However, the ignored semantic information can be vital for some fuzzing heuristics. For example, TortoiseFuzz [32] analyzes potentially vulnerable functions in the target program and calculates a new metric coverage count to guide seed scheduling. If TortoiseFuzz fuzzes the example code in Fig. 1, it will ideally spend more time on the dangerous function `xmalloc`. Unfortunately, it cannot notice the potential bug when fuzzing the bug pattern summarized from the example code. In other words, the bug pattern is not completely fair for fuzzers that rely on semantic information. These fuzzers may get worse performance than they should.

Nonetheless, we aim to defend this limitation by presenting two perspectives. Firstly, ShadowBug's strength lies in its ability to refine existing synthetic benchmarks for bug-triggering, reducing the risk of overfitting. This development represents a step closer to practical synthetic benchmarks. In fact, most advanced fuzzers seek to enhance coverage by addressing explicit constraints, rendering previous synthetic benchmarks increasingly inapplicable. However, ShadowBug can remain impartial for fuzzers that have no strong utilization of semantic information. Secondly, we can enhance the bug pattern to maintain some level of semantic information. For

example, we can retain function calls, but without context, as demonstrated by a code statement that calls `xmalloc` with an argument of 0 in the code samples.

VII. RELATED WORKS

As previously mentioned, fuzzing benchmarks fall into two categories: real-world CVE benchmarks and synthetic benchmarks. One popular CVE benchmark is the Google Fuzzer Test-Suite [33], which comprises carefully selected bugs from popular open-source software (OSS) projects. This benchmark has been widely adopted in recent years and has been further enhanced to create the FuzzBench [34] fuzzer evaluation platform. UNIFUZZ [14] is another platform that evaluates fuzzers on a large scale using a multi-angle analysis of fuzzer performance with different metrics like bug-finding time, coverage, and bug quality. Another benchmark is Magma [35], which collects CVEs from seven target projects. It addresses CVEs from the same OSS but with different patches by porting the old bugs to updated versions and integrating them into one binary, making the benchmark have a dense bug distribution. DARPA Cyber Grand Challenge (CGC) [36] is a set of manually designed bugs, which is slightly different from real-world CVE benchmarks. The bugs are usually copies of known CVE bugs, but they are introduced by choice. However, as previously mentioned in Section I, real-world CVE benchmarks do not conflict with ShadowBug. It generates advanced synthetic bugs that can make up for the weakness of real-world CVE benchmarks.

LAVA [3] is a popular synthetic benchmark that has been widely evaluated in past works [10], [11], [37], [38], [39], [40], [41], [42]. LAVA performs dynamic taint analysis on target programs and constructs bugs from inactive variables at branch-taking. It then inserts explicit out-of-bounds access protected by magic-byte comparisons. LAVA only uses explicit constraints, which are outdated and easy to bypass, and thus insufficient and unrealistic in modeling real bugs. Instead, ShadowBug is aware of the implicit constraints and designs the transformation from normal constraints to their implicit form, which complements the drawbacks of previous works.

FixReverter [43] is a bug-insertion method. It identifies common syntax patterns of CVE bug fixes, matches them with the target program, and reverses the compatible bug fixes with respect to the context to reproduce the bugs in the source code. Different from ShadowBug, the bug patterns of FixReverter contain semantic information about CVE bugs. The strength is that it supports all sorts of bug-triggering mechanisms. However, as CVE bugs can be complicated, they are not able to manage the bugs, e.g., modifying the solution space of the bugs. The benchmark is close to the real-world CVE benchmark. In contrast, ShadowBug is more flexible at editing bugs, generating PoC and guaranteeing the feasibility of each bug.

EvilCoder [2] is also a bug-injection method that performs static taint tracking on source code to detect potentially dangerous memory operations on tainted variables, such as `memcpy` and `strncpy`. It removes input validation of the candidate memory operations to inject the bugs. However, it has no guarantee of bug-triggering, and the bug will only be triggered if invalid memory access occurs with unprotected inputs. Compared with ShadowBug, EvilCoder relies on the existing statements in the source code and is inappropriate to support large-scale bug generation.

Apocalypse [1] is another bug-insertion method that constructs an Error Transition System (ETS) based on state transitions in different CFG nodes along a path. A bug is triggered if the final state is reached, and the state transitions are synthesized by symbolic execution to ensure the feasibility of the inserted bug. However, the constraints used in Apocalypse are still explicit, which benefits fuzzing heuristics focused on constraint-solving. Not to mention the state transitions are constructed from symbolic execution, which can even bias the related fuzzers [11], [38], [42], [44]. On the contrary, ShadowBug generates bugs that better simulate real-world bugs with implicit constraints which are fairer to all fuzzing heuristics. The implicit constraints more realistically simulate the challenges and complexity of finding bugs in real-world software than Apocalypse.

VIII. CONCLUSION

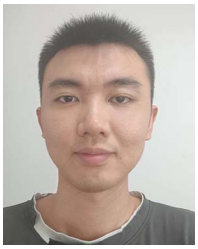
In this paper, we present ShadowBug, a novel methodology for generating synthetic bugs that enhance existing benchmarks in three key ways. First, it ensures a desired distribution of bugs throughout the target program based on a metric exploration difficulty that specifies the expected fuzzing trials needed to bypass each branch. Second, ShadowBug addresses the issue of lacking implicit constraints in current benchmarks by implementing a transformation from normal inequalities to their implicit forms. The inserted bugs are then quantified by the metric exploitation difficulty. Finally, ShadowBug combines exploration and exploitation difficulties into a new metric called "bug difficulty", which allows for a more fine-grained evaluation of fuzzers.

To demonstrate the effectiveness of ShadowBug, we created a synthetic benchmark comprised of eight programs from Binutils and Google Fuzzer Test-Suite with a total of 466 bugs. We tested this benchmark against five established fuzzers and found that 391 out of 466 bugs were triggered. We also verified the feasibility of idle bugs through automatically generated PoCs. In summary, ShadowBug creates synthetic bugs with distinct difficulty classifications, which enables more realistic simulations of real-world bugs. It further provides a more detailed description of the strengths of different fuzzing heuristics regarding exploring new branches and triggering the bugs.

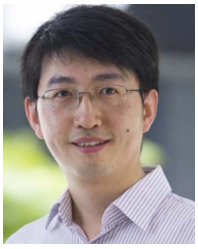
REFERENCES

[1] S. Roy, A. Pandey, B. Dolan-Gavitt, and Y. Hu, "Bug synthesis: Challenging bug-finding tools with deep faults," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2018, pp. 224–234.
[2] J. Pewny and T. Holz, "EvilCoder: Automated bug insertion," in *Proc. 32nd Annu. Conf. Comput. Secur. Appl.*, 2016, pp. 214–225.
[3] B. Dolan-Gavitt et al., "LAVA: Large-scale automated vulnerability addition," in *Proc. IEEE Symp. Secur. Privacy*, 2016, pp. 110–121.

[4] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 2123–2138.
[5] M. Boehme, C. Cadar, and A. Roychoudhury, "Fuzzing: Challenges and reflections," *IEEE Softw.*, vol. 38, no. 3, pp. 79–86, May/Jun. 2021.
[6] Z. Zhang, W. You, G. Tao, Y. Aafer, X. Liu, and X. Zhang, "StochFuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting," in *Proc. IEEE Symp. Secur. Privacy*, 2021, pp. 659–676.
[7] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *Proc. USENIX Workshop Offensive Technol.*, 2020, pp. 1–12.
[8] D. She, A. Shah, and S. Jana, "Effective seed scheduling for fuzzing with graph centrality analysis," in *Proc. IEEE Symp. Secur. Privacy*, 2022, pp. 2194–2211.
[9] J. Liang et al., "PATA: Fuzzing with path aware taint analysis," in *Proc. IEEE Symp. Secur. Privacy*, 2022, pp. 1–17.
[10] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: Fuzzing with input-to-state correspondence," in *Proc. Netw. Distrib. System Secur. Symp.*, 2019, pp. 1–15.
[11] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM : A practical concolic execution engine tailored for hybrid fuzzing," in *Proc. USENIX Secur. Symp.*, 2018, pp. 745–761.
[12] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proc. IEEE Symp. Secur. Privacy*, 2018, pp. 711–725.
[13] "Common vulnerability scoring system (CVSS)," 2023. Accessed: Jan. 22, 2023. [Online]. Available: https://nvd.nist.gov/vuln-metrics/cvss
[14] Y. Li et al., "UNIFUZZ: A holistic and pragmatic metrics-driven platform for evaluating fuzzers," in *Proc. USENIX Secur. Symp.*, 2021, pp. 2777–2794.
[15] "Undefined behavior sanitizer with clang," 2023. Accessed: Jan. 25, 2023. [Online]. Available: http://clang.llvm.org/docs/UndefinedBehaviorSanitizer
[16] J. Jiang, L. Chen, X. Wu, and J. Wang, "Block-wise abstract interpretation by combining abstract domains with SMT," in *Proc. Verification, Model Checking, Abstract Interpretation*, 2017, pp. 310–329.
[17] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017.
[18] H. Chen et al., "Hawkeye: Towards a desired directed grey-box fuzzer," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 2095–2108.
[19] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida, "ParmeSan: Sanitizer-guided greybox fuzzing," in *Proc. USENIX Secur. Symp.*, 2020, pp. 2289–2306.
[20] H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang, "Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction," in *Proc. IEEE Symp. Secur. Privacy*, 2022, pp. 1613–1627.
[21] P. Zong, T. Lv, D. Wang, Z. Deng, R. Liang, and K. Chen, "FuzzGuard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning," in *Proc. USENIX Secur. Symp.*, 2020, pp. 2255–2269.
[22] S. Nagy and M. Hicks, "Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing," in *Proc. IEEE Symp. Secur. Privacy*, 2019, pp. 787–802.
[23] M. Zalewski, "American fuzzy lop," 2019. Accessed: Aug. 14, 2023. [Online]. Available: http://lcamtuf.coredump.cx/afl
[24] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, "BEACON: Directed grey-box fuzzing with provable path pruning," in *Proc. IEEE Symp. Secur. Privacy*, 2022, pp. 36–50.
[25] Y. Li, A. Albarghouthi, Z. Kincaid, A. Gurfinkel, and M. Chechik, "Symbolic optimization with SMT solvers," in *Proc. Annu. ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, 2014, pp. 607–618.
[26] "Libtooling: A library to support writing standalone tools based on clang," 2023. Accessed: Jan. 25, 2023. [Online]. Available: https://clang.llvm.org/docs/LibTooling.html
[27] "Intel-pin: A dynamic binary instrumentation tool," 2020. Accessed: Oct. 25, 2020. [Online]. Available: https://software.intel.com/sites/landingpage/pintool/docs/81205/Pin/html/
[28] U. S. Barbara and SEFCOM, "Angr: Python framework for analyzing binaries," 2020. Accessed: Oct. 23, 2020. [Online]. Available: http://angr.io/
[29] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 1032–1043.
[30] R. Swiecki, "Honggfuzz," 2020. Accessed: Aug. 14, 2023. [Online]. Available: https://github.com/google/honggfuzz
[31] "Circumventing fuzzing roadblocks with compiler transformations," 2023. Accessed: Jan. 25, 2023. [Online]. Available: https://lafintel.wordpress.com/
[32] Y. Wang et al., "Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020, pp. 1–17.
[33] "Google fuzzer test suite," 2022. Accessed: Mar. 12, 2022. [Online]. Available: https://github.com/google/fuzzer-test-suite
[34] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "FuzzBench: An Open Fuzzer Benchmarking Platform and Service," in *Proc. ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Foundations Softw. Eng.*, 2021, pp. 1393–1403.
[35] A. Hazimeh, A. Herrera, and M. Payer, "MAGMA: A ground-truth fuzzing benchmark," in *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 4, pp. 1–29, 2020.
[36] GrammaTech, "Grammatech blogs: The cyber grand challenge," 2016. [Online]. Available: https://blogs.grammatech.com/the-cyber-grand-challenge
[37] H. Peng, Y. Shoshitaishvili, and M. Payer, "Tfuzz: Fuzzing by program transformation," in *Proc. IEEE Symp. Secur. Privacy*, 2018, pp. 697–710.
[38] Y. Chen et al., "SAVIOR: Towards bug-driven hybrid testing," in *Proc. IEEE Symp. Secur. Privacy*, 2020, pp. 1580–1596.
[39] P. Chen, J. Liu, and H. Chen, "Matryoshka: Fuzzing deeply nested branches," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 499–513.
[40] W. You et al., "ProFuzzer: On-the-fly input type probing for better zero-day vulnerability discovery," in *Proc. IEEE Symp. Secur. Privacy*, 2019, pp. 769–786.
[41] C. Lyu et al., "MOPT: Optimized mutation scheduling for fuzzers," in *Proc. USENIX Secur. Symp.*, 2019, pp. 1949–1966.
[42] N. Stephens et al., "Driller: Augmenting fuzzing through selective symbolic execution," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2016, pp. 1–16.
[43] Z. Zhang, Z. Patterson, M. Hicks, and S. Wei, "FIXREVERTER: A realistic bug injection methodology for benchmarking fuzz testing," in *Proc. USENIX Secur. Symp.*, 2022, pp. 3699–3715.
[44] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2008, pp. 209–224.



ZHENGXIANG ZHOU received the B.E. degree in computer science and technology from The Chinese University of Hong Kong, Shenzhen, China, in 2019. He is currently working toward the Ph.D. degree with the Department of Computer Science, City University of Hong Kong, Hong Kong. His research interests include fuzzing and software engineering.



CONG WANG (Fellow, IEEE) is currently a Professor with the Department of Computer Science, City University of Hong Kong, Hong Kong. His research interests include data and network security, blockchain and decentralized applications, and privacy-enhancing technologies. He was the co-recipient of the IEEE INFOCOM Test of Time Paper Award 2020, Best Paper Award of IEEE ICDCS 2020, ICPADS 2018, and MSN 2015, and Best Student Paper Award of IEEE ICDCS 2017. At CityU, he was the recipient of the Outstanding Researcher Award in 2019, Outstanding Supervisor Award in 2017, and President's Awards in 2016 and 2019. He is a founding Member of the Young Academy of Sciences of Hong Kong and Research Fellow of the Hong Kong Research Grants Council. He is the Editor-in-Chief of IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING.