VALL-NUT: Principled Anti-Greybox-Fuzzing

Yuekang Li[1,2] Guozhu Meng[3,4*] Jun Xu[5] Cen Zhang[1,2] Hongxu Chen[1] Xiaofei Xie[1]
Haijun Wang[6*] Yang Liu[1]
[1]Nanyang Technological University [2]Continental-NTU Corporate Lab
[3]SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences
[4]School of Cyber Security, University of Chinese Academy of Sciences
[5]Stevens Institute of Technology [6]Xi'an Jiaotong University

***Abstract*—Greybox fuzzing is a widely used technique for software testing that has been adopted by practitioners and researchers to disclose a great number of vulnerabilities in various software. However, adversaries also weaponize greybox fuzzing to mine vulnerabilities for malicious intentions. This poses considerable threats to software systems.**

To counteract the misuse of greybox fuzzing, we propose VALL-NUT, a novel approach to harden software with properties to combat greybox fuzzing. We dissect the major strategies that facilitate the success of greybox fuzzing, and accordingly propose three types of neutralizing schemesseed queue explosion, seed attenuation, and feedback contamination. We evaluate VALL-NUT against the mainstream greybox fuzzers on multiple real-world benchmark programs. The results show that VALL-NUT can reduce an average of 34% code coverage and 76% detected crashes in 24-hour tests. Moreover, we conduct comparisons with two recent studies which show VALL-NUT can achieve a superior deduction of detected crashes.

***Index Terms*—Software testing, Fuzzing, Anti greybox fuzzing**

I. INTRODUCTION

Since the introduction in the early 1990s, fuzzing [1] has become one of the most effective approaches for software testing. The philosophy of fuzzing is to run the target program with random inputs and watch for abnormal behaviors. Such abnormal behaviors often indicate the presence of defects.

Based on the awareness of structural knowledge on the target program, fuzzing techniques are usually categorized as blackbox, whitebox and greybox. Blackbox fuzzers (*e.g.*, [2]) deem the target program as a blackbox and generate inputs following mathematical or statistical models. The lack of knowledge about the internals of target programs limits the utility of such fuzzers. Whitebox fuzzers (*e.g.*, [3]–[5]), on the contrary, have full access to the target program and can perform heavy program analysis such as symbolic execution to gain more profound comprehension. However, whitebox fuzzers are less scalable because of the expensive program analysis. Greybox fuzzers strike a balance between effectiveness and scalability. Greybox fuzzers leverage light-weight program instrumentation to capture execution feedback from the target program under test. The feedback can effectively aid the generation of high-quality seeds while barely decreasing the execution speed.

[*]Corresponding authors: Guozhu Meng and Haijun Wang

Being effective yet scalable, greybox fuzzing has become the mainstream technique for vulnerability detection. For instance, AFL [6], one of the most popular greybox fuzzers, has assisted the discovery of thousands of vulnerabilities. While greybox fuzzing benefits security practitioners and researchers, unsurprisingly, it is also used by adversaries to mine zero-day vulnerabilities for malicious intentions. In this sense, greybox fuzzing turns into a threat against software systems that undergo insufficient tests. To mitigate this threat, an intuitive reaction is to "sufficiently" extend greybox fuzzing on software prior to release. This, however, is often infeasible due to constrained time budgets. Even worse, they lack theoretical foundations to determine "sufficiency" of fuzz testing.

Instead of racing with adversaries by running fuzzing, we turn to the strategy of anti-greybox fuzzing. The idea is to harden the software with properties that can combat greybox fuzzing. There are two nontrivial challenges for achieving this goal. On the one hand, we should enable the software to thwart the fundamental mechanisms of grey-box fuzzing, instead of exploiting the weaknesses in *the implementations of specific tools*. Anti-fuzzing techniques that follow the latter strategy (*e.g.*, [7], [8]) are fragile against varieties and advances in implementations. On the other hand, anti-fuzzing techniques have to be non-intrusive to the target software. More specifically, we need to preserve the original semantics and avoid downgrading the execution efficiency for normal usage.

In this study, we propose VALL-NUT[1] to address the aforementioned challenges. Specifically, we first summarize a systematic model of grey-box fuzzing, which captures the major contributing strategies and abstracts away implementation details. This model reveals that the success of greybox fuzzing is largely attributed to a feedback system powered by three key components, namely the *seed evaluator*, the *seed mutator* and the *feedback collector*. To subvert this feedback system, VALL-NUT generates *fuzzing obstacle code* and implants it into the target program. These pieces of fuzzing-obstacle code sabotage the feedback system from three aspects–*seed queue explosion*, *seed attenuation* and *feedback contamination*. More details about these three techniques can be found at Section IV. Owing to these counter measurements, VALL-NUT can effectively neutralize the power of greybox fuzzers.

[1]The term "VALL-NUT" derives from the defensive plant "WALL-NUT" in the game of *Plants vs. Zombies*, which acts as a shield for the player's other plants due to its high durability.

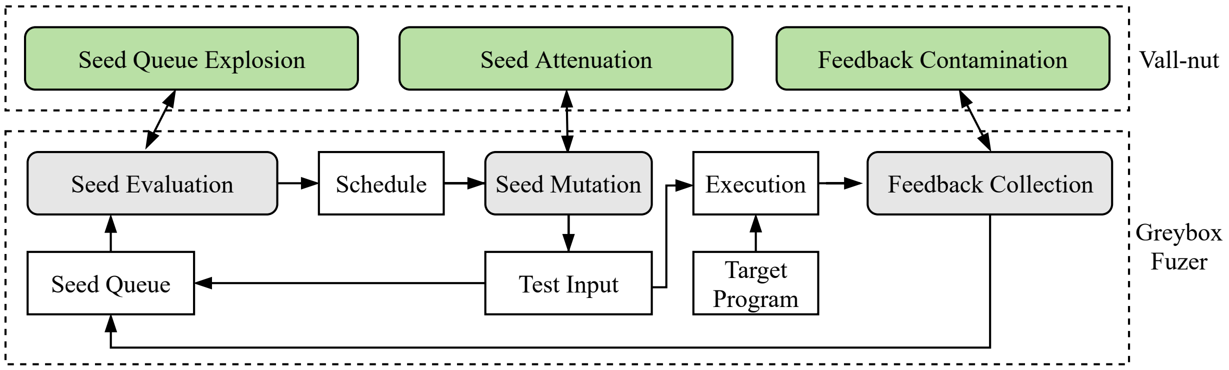


Fig. 1: The model of Greybox Fuzzing and VALL-NUT

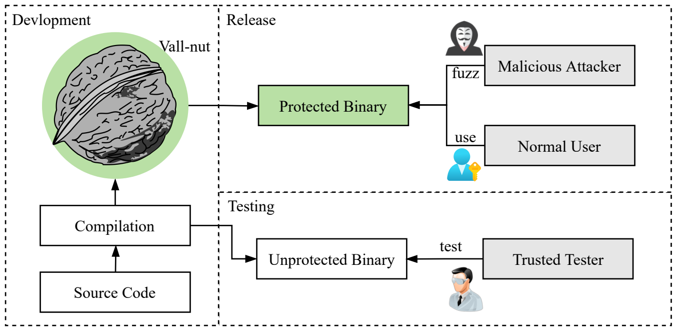


Fig. 2: Application Scenario of VALL-NUT

Moreover, these techniques are carefully designed and used to avoid hurting the normal execution, in order to address the second challenge.

To evaluate the effectiveness and efficiency of VALL-NUT, we conduct extensive experiments scaling up to more than 2,400 CPU hours on 6 real-world programs. Our evaluation results demonstrate that VALL-NUT can reduce 34% code coverage and 76% detected crashes on average in 24 hours of fuzzing. Particularly, we find that VALL-NUT fundamentally restrains the overall performance of greybox fuzzing, making the latter even less effective than blackbox fuzzing. Last but not least, VALL-NUT shows higher expressiveness than two state-of-the-art anti-fuzzing techniques [9], [10].

Contributions. We summarize the contributions as follows.

- We systematically study the strengths of greybox fuzzing and propose counteractive strategies accordingly.
- We tailor the strategies into VALL-NUT, an anti-greybox-fuzzing system that can effectively restrain the performance of greybox fuzzing.
- We implement VALL-NUT and evaluate VALL-NUT with real-world programs. The evaluation results show that VALL-NUT can significantly hinder the program coverage and bug finding capabilities of greybox fuzzing.

II. APPLICATION SCENARIO

This research focuses on anti-greybox-fuzzing techniques. To better clarify our scope, we define greybox fuzzing as follows: fuzzing techniques that can leverage on light-weight runtime *feedback* to keep interesting inputs as *seeds* for future *mutation*. We also model greybox fuzzing in Figure 1 and discuss it in detail in Section III-A.

At the high level, we focus on developing techniques that apply to the application scenario presented in Figure 2. *Prior to release of the target software*, the vendor applies our technique to the source code and produces a self-protecting binary. This binary can then be released to the public. In this research, we assume the adversaries can acquire the released binary but have no access to the source code. Moreover, VALL-NUT is applied only for the released binary and does not affect the internal testing process. It provides protection against adverse greybox fuzzing in cases where the trusted testers failed to detect all the bugs before releasing the software.

Under the above application scenario and assumptions, we expect the binary hardened by our techniques to achieve three goals: First, running greybox fuzzing against the protected binary would not gain more than using blackbox fuzzing. We believe this is a reasonable bound, since higher anti-greybox-fuzzing effectiveness would simply push the adversaries to adopt blackbox fuzzing; Second, the binary has robust anti-fuzzing properties, meaning that it will render similar expressiveness against different implementations of grey-box fuzzing; Third, the binary neither interrupts normal functionality nor significantly reduces the execution efficiency. Failing to satisfy these requirements would prevent vendors from using our techniques.

III. OVERVIEW

A. Greybox Fuzzing Model

In Figure 1, we point out the 3 key components of greybox fuzzers – *the seed evaluator*, *the seed mutator* as well as *the feedback collector*. ❶ The seed evaluator is the component to facilitate the maintenance of the seed queue. Typically, a seed evaluator is in charge of selecting the next seed to mutate and propose how much computational power to be spent on that seed (aka "power schedule" in [11]). A good seed evaluator can help maximize the output via providing optimal solutions for seed selection and power scheduling. A tampered seed evaluator may wrongly evaluate the quality of the seeds or even starve certain seeds. ❷ The seed mutator is the bread and butter for all kinds of fuzzers, not just greybox fuzzers. A good seed mutator can effectively generate new inputs with good quality. A tampered seed mutator may not be able to generate new inputs that are "essentially different" from the seed. By "essentially different", we mean not only the content but also the program coverage of the inputs are different. ❸ The feedback collector is the component to collect the feedback from the instrumentation of the target program during the execution. A good collector can report accurate and compact feedback. A tampered collector may report wrong and confusing feedback.

B. Overview of Our Approach

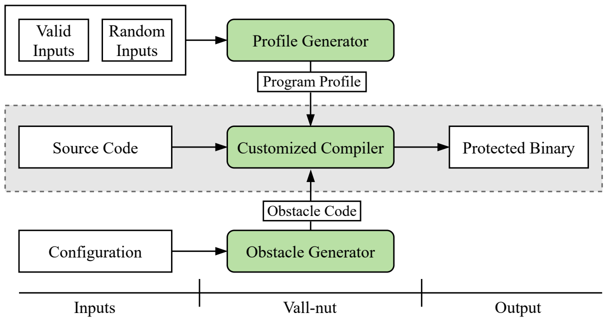


Fig. 3: Workflow of VALL-NUT

As shown in Figure 3, VALL-NUT first profiles the target program to measure the hitting frequency of basic-block transitions. This profiling, in particular the frequency information, helps identify some key locations – Code that is frequently exercised by fuzzers but rarely hit in normal execution. These key locations typically reside in the error handling logic of the target program. At each site of those locations, VALL-NUT then plants fuzzing-obstacle code in accordance with the developers' specifications. These pieces of fuzzing-obstacle code, while getting fuzzed, would impede the feedback mechanism that powers greybox fuzzing. More specifically, the fuzzing-obstacle code ❶ drives greybox fuzzing to keep the seeds with lower quality, thus disorders the *seed evaluator*; ❷ leads greybox fuzzing to generate and maintain seeds with increasing size, making *seed mutator* less effective; ❸ contaminates the feedback to greybox fuzzing using schemes such as overwhelming the feedback storage or introducing misleading feedback.

IV. THE VALL-NUT METHODOLOGY

A. Profile Generator

In VALL-NUT, we first generate the edge-level frequency profile for the target program. We then use this profile result to identify error handling paths for injecting the fuzzing obstacle code. By error handling paths, we refer to the paths executed by invalid inputs. Previous work uses basic-block-level frequency to identify error handling paths [12]. However, we use an edge-level frequency because the program may not really have error handling code. For example, for program 1 in Figure 4, there are four basic blocks A, B, C and D. At A, the program decides whether the input is valid or not. If the input is valid,

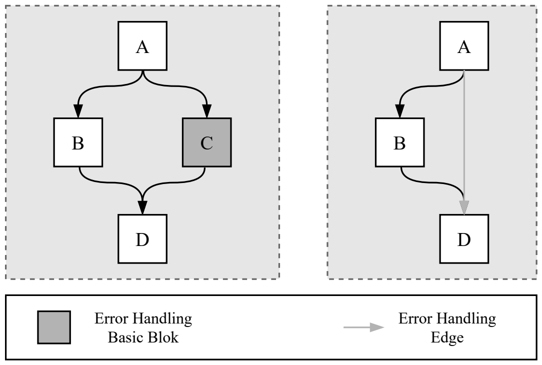


Fig. 4: Demonstration of error handling path

then the control flow goes into B; otherwise C. In this case, we can say the error handling basic block is C. However, sometimes the program simply does not handle errors. Like program 2 in Figure 4, the program handles the valid input with B and if the input is invalid, the control flow directly goes from A to D and ends the execution. In this case, while we cannot find the error handling basic block, we can still use the edge between A and D to represent the error handling path. Thus, we use *edge-level frequency profile*.

The purpose of performing injection on error handling paths is twofold. First, we attempt to reduce the effect of VALL-NUT on normal usage as much as possible. The intuition is that normal users tend to supply *valid inputs* to the software, while fuzzers supply *invalid inputs* to the software most of the time. Thus, injecting at error handling paths can hinder the fuzzers without affecting normal usage much. Second, error handling paths injection can help with the *seed queue explosion* strategy by misleading the fuzzer to keep a lot of invalid inputs as seeds in queue. A detailed discussion about this is in Section IV-B.

From Figure 3, we can see the profile generator requires the developer to provide a set of valid inputs V and a set of random inputs R. First, we execute the original program binary with the valid inputs in V and collect the set of executed edges $E(V)$. After that, we execute the program with random inputs in R and collect the set of executed edges $E(R)$. Then we classify the *error handling edges*(EHE) as edges executed by at least $n\%$ of the random inputs and not executed by any valid input:

$$\begin{aligned} EHE = &\{e : \forall i{\in}\mathcal{P}(R),\ s.t. \, |i| > |R| * n/100, \\ & e{\in}E(i) \, and \, e{\notin}E(V)\} \end{aligned} \tag{1}$$

where $\mathcal{P}(E(R))$ is the power set of $E(R)$ and $E(i)$ is the set of executed edges with inputs from i. The value of n in Equation 1 works as a throttle for controlling the number of instrumented locations. A higher value of n means that a smaller number of edges will be included in EHE.

Later, as shown in Figure 3, the customized compiler will break up the edges in EHE and insert the obstacle code for hindering greybox fuzzers. For example, after injecting the obstacle code, the sample program in Listing **??** will become equivalent to the program shown in Listing **??**.

B. Fuzzing Obstacle Generator

In VALL-NUT, we use the fuzzing obstacle generator to generate the fuzzing obstacles which are to be injected by breaking up the error handling edges identified through profiling. The fuzzing obstacles are specially designed code snippets for carrying out the three strategies proposed in Figure 1.

Algorithm 1: The Fuzzing Obstacle Code

```
   Input: input: test input for the program;
          jump_limit: the limit of statement jumps
 1 i ← 0;
 2 initialize input_length_steps[];
 3 // get the length of the input
 4 input_len ← length(input);
 5 // initiate the labels for the jump tables
 6 jt1labels ← [jt1label1, jt1label2, ... jt1labeln, ... jt1endlabel];
 7 jt2labels ← [jt2label1, jt2label2, ... jt2labeln, ... jt2endlabel];
 8 ...
 9 jtklabels ← [jtklabel1, jtklabel2, ... jtklabeln, ... jtkendlabel];
10 jt1label1:
11 if i < jump_limit then
12     i ← i + 1;
13     rand_index ← rand(0, length(jt1labels));
14     goto jt1labels[rand_index];
15 else
16     goto jt1endlabel;
17 ...
18 jt1labeln:
19 if i < jump_limit then
20     i ← i + 1;
21     rand_index ← rand(0, length(jt1labels));
22     goto jt1labels[rand_index];
23 else
24     goto jt1endlabel;
25 ...
26 jt1endlabel:
27 if input_len > input_lengths_steps[0] then
28     i ← 0;
29     goto jt2label1;
30 else
31     goto jtkendlabel;
32 jt2label1:
33 ...
34 jt2endlabel:
35 if input_len > input_lengths_steps[1] then
36     i ← 0;
37     goto jt3label1;
38 else
39     goto jtkendlabel;
40 ...
41 jtkendlabel:
42 // this is the end of the obstacle code
43 // just go back to the original program logic
```

Algorithm 1 shows the logic of the fuzzing obstacle code. The code is generated by the *fuzzing obstacle generator* according to the user configuration and the configurable variables are *jump_limit*, *input_length_steps[]* and the size of each jump table. The core logic of the obstacle code is a cluster of collaborative jump tables. A jump table is a method of transferring program control from one part to another via jumping between labeled locations in the code. Since the labels are stored in a table, the program control can be transferred to any labeled location by querying the table. Line 6 to line 9 in Algorithm 1 are examples of jump table declarations. In general, the obstacle code consists of two different types of jump tables. The first type of jump table is *jt1* in Algorithm 1. This jump table contains more labels and mainly helps with the *seed queue explosion* strategy. The second type of jump tables are *jt2* to *jtk* in Algorithm 1. These jump tables help with the *seed attenuation* strategy. After the jump tables are declared, the labels are put into the code to mark the locations for jumping, like line 10 or line 18. With the jump tables, we have three decisions to make. ❶ While jumping between the labeled locations, we need to tell when to break out. This is controlled by the *jump_limit* variable, which limits the number of jumps before exiting the jump table logic. As shown in line 11 and line 19, the program breaks out from a jump table if the jump number limit is reached. ❷ After settling the problem of when to stop jumping, we need to decide how to select the next location for jumping. In VALL-NUT, the next location to jump to is selected randomly from the label table as shown in line 21 and line 22. ❸ Since we have multiple jump tables, we also need to decide when or whether the program should move on to the next jump table. In VALL-NUT, the program will continue to execute the next jump table only if the length of the input is larger than certain value as shown with the `if` blocks in line 27 and line 35. This mechanism makes larger inputs cover more jump tables. Note that the sizes of jump tables 2 to k are the same and are much smaller than the size of jump table 1 to reduce the size of the obstacle code.

Here we explain the rationale behind the three strategies shown in Figure 1 and how the aforementioned fuzzing obstacle code can help to carry out the strategies in detail.

Seed Queue Explosion. Some existing advances in greybox fuzzing, like AFLFast [13], FairFuzz [14] or Vuzzer [12] propose to degrade the importance of the seeds executing error handling paths. The intuition is that such inputs are often far from the major working logic of the program where the bugs are more likely to reside. Reversely, VALL-NUT can explode the seed queue maintained by the greybox fuzzers by misleading them to keep a large amount of inputs exercising error handling paths.

Since the program can jump freely from one labeled location to another, a jump table containing n labels can contribute n basic-blocks and n^2 basic-block transitions (edges). This means that the injected jump tables can provide a considerable amount of program coverage feedback to trick the fuzzer to keep a lot of inputs as seeds exercising the injected code. Furthermore, as the fuzzing obstacle code is injected in the error handling paths, the new seeds are kept since the injected code are invalid inputs. Without the fuzzing obstacle code, greybox fuzzers may *keep only a few (chaff) inputs* for each error handling path. Meanwhile, with the jump table 1 in the fuzzing obstacle code, greybox fuzzers will mistakenly keep a lot more inputs exercising the error handling paths. As a result, greybox fuzzers will waste a lot of computational power on seeds with worse quality.

Seed Attenuation. GBFs prefer more compact seeds. The intuition is that mutating smaller seeds are more likely to hit the *interesting bytes* and cover new paths in the target program. For example, if a fuzzer is mutating an XML file to test an XML parser, then mutating the tags is more important than mutating the data and mutating a smaller XML file is more likely to hit the tags. In this sense, some GBFs are trying to trim the seeds by deleting certain bytes that do not affect program coverage. For example, Fairfuzz employs aggressive trimming strategies to bring down the size of the seeds as much as possible. Reversely, VALL-NUT can guide the GBF to keep larger and larger inputs as seeds, in which the *interesting bytes* are attenuated.

In VALL-NUT, *seed attenuation* is achieved by providing more feedback when the seed size is large. In Algorithm 1, we can see that the jump tables 2 to k inside the fuzzing obstacle code provides more feedback when the seed size gets larger. In the actual implementation, we have additional instrumentation to hook the input reading location of the original program and keep a copy of the input length for the fuzzing-obstacle code. The $input\_lengths\_steps[]$ variable supplied by the developers works as the staircase to guide greybox fuzzing to keep larger and larger seeds step by step. As a result, greybox fuzzers keep more seeds with larger sizes and mutate less effectively.

Feedback Contamination. One important feature of greybox fuzzing is that they collect runtime feedback to identify interesting inputs and keep them as seeds for further mutations. However, they are faced with two problems. ❶ A greybox fuzzer cannot hold infinite feedback information. Thus, the buffer for holding this information can get saturated, which limits the scalability. In fact, CollAFL [15] addressed this issue and proposed an algorithm for resolving the feedback record collision problem for normal size programs. ❷ The other problem is that some inputs can generate different execution traces across different runs because of multi-threading or random logic used in the program. This can cause confusions to a fuzzer when it tries to store the feedback collected from executing such inputs. Taking AFL as an example, it performs more calibration runs [16] for seed inputs that behave differently across runs or otherwise the feedback of such inputs will contaminate the existing feedback record kept by the fuzzer. Thus, more computation power is spent to calibrate such inputs and relatively less computation power is used for mutation and other stuff. With these observations, VALL-NUT contaminates the feedback records of greybox fuzzers by saturating the feedback records and introducing inputs with variable behaviours.

In VALL-NUT, the jump tables provide greybox fuzzers with a large amount of dummy feedback, which can saturate the feedback records very quickly. Other than this, the next location to jump to is selected randomly from the jump table. Thus, the inputs triggering the jump tables naturally behave differently across runs. To further disguise these inputs with the inputs executing normal paths, we add instrumentation that contains program irrelevant random logic to the place where we hook the input reading. The effect is that all inputs hold different execution traces across different runs and the inputs exercising the fuzzing obstacle code behave very unstably from fuzzers' point of view. As a result, greybox fuzzers cannot make use of the feedback effectively when fuzzing the program protected by VALL-NUT as it is contaminated.

V. EVALUATION

Our tool VALL-NUT is implemented in Python, C, and C++, using Radamsa [17] to generate random inputs for profile generation, LLVM [18] for both the profiling instrumentation and the injection of the fuzzing obstacle code, and Redis [19] as the profile instrumentation log database.

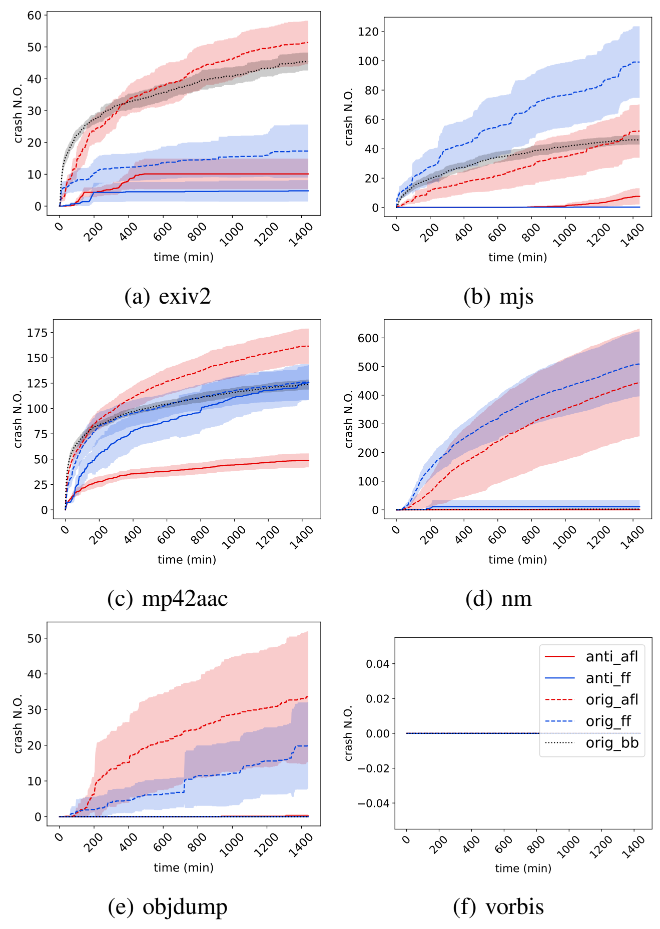
A. Evaluation Setup

Infrastructure. The experiments are conducted on 2 machines, each with Intel(R) Core(TM) i9-7940X CPU @ 3.10GHz, 28 logical processor cores, and 128GB RAM. The experiment for each target on each fuzzer is repeated 10 times to mitigate the randomness of fuzzing. Our experiments scale up to more than 2400 CPU hours.

Evaluated Fuzzers. We evaluate our techniques on 3 different fuzzers, including AFL-QEMU, AFL-Blackbox (AFL-BB), Fairfuzz-QEMU. ❶ AFL-QEMU is AFL running in *QEMU mode* for fuzzing program binaries. It utilizes QEMU to collect the coverage feedback of the target program. AFL is the baseline fuzzer used in many previous researches and it is recognized as the state-of-the-art greybox fuzzer. ❷ AFL-BB is a modified version of AFL-QEMU which is the same as AFL-QEMU except that its seed queue will always loop over the initial seeds. In short, AFL-BB is the blackbox counterpart of AFL-QEMU. AFL-BB can represent blackbox fuzzers with reasonably good mutation operators. Although VALL-NUT targets on greybox fuzzing but not blackbox fuzzing, we still run AFL-BB to check if VALL-NUT can limit the performance of greybox fuzzers to be equal to or even worse than the blackbox fuzzer. ❸ Fairfuzz-QEMU uses the *QEMU mode* of the current state-of-the-art greybox fuzzer FairFuzz [14] which targets on low-frequency edges. FairFuzz is considered as an improved version of AFL and it used to check if the greybox fuzzers with awareness of the path frequency can evade our strategy of inserting fuzzing obstacles on high frequency paths.

Evaluated Programs. Our criteria for choosing the programs are: 1) the programs should be diverse in program sizes; 2) the programs should be diverse in their functionality. 3) the programs should be used in previous works or benchmark sets. With these criteria, we chose 6 programs for evaluation and Table I shows the details about the target programs. Additionally, the initial seeds we use for fuzzing either align with the previous works or from the default seed corpus provided by AFL.

Performance Metrics. Following Klees et al.'s [25] recommendation, we use the number of crashes and edge level coverage as measurements for the fuzzing results. We also apply statistical tests [26] to the experimental results. We use the Mann Whitney



(a) exiv2 (b) mjs (c) mp42aac (d) nm (e) objdump (f) vorbis

Fig. 5: Number of Detected Crash Over Time (Lower is better)

TABLE I: Details of Evaluated Programs

Target	Previous Work	Bin Size	Input Format
exiv2	[20] [21]	4830K	jpg
mjs	[22]	156K	javascript
mp42aac	[20]	1877K	mp4
nm	[14] [13]	5055K	elf
objdump	[14] [20]	7408K	elf
vorbis	[23] [24]	321K	ogg

U-test [27] to determine the statistical significance of the results, and the $\hat{A}_{12}$ statistics [28] to show the possibility that a fuzzer performs better on the original binary than on the protected binary according to all runs.

VALL-NUT Configuration. For selecting the error handling edges, we set the n in Equation 1 to be 95. For the fuzzing obstacle code, we use the $jump\_limit$ of 10, which means each input exercising the fuzzing obstacle code occupies 10 edges in the jump table. The $input\_length\_steps$ we use starts from 100 and ends at 8000 with each step of 5, which means VALL-NUT will try to guide the fuzzers to eventually keep seed inputs larger than 8000 bytes. The jump table used for *seed queue explosion* (*jt1labels* in Algo. 1) contains 10k labels. The jump tables used for *seed attenuation* (*jt2labels* to *jtklabels* in Algo. 1) contains 2 labels each.

In addition, the compiled binary of the fuzzing obstacle code is about 1.2 MB, resulting in the protected binaries 1.2 MB larger than its original counterpart.

Research Questions. With the experiments, we aim to answer the following research questions:

RQ1. How well can VALL-NUT hinder bug finding?
RQ2. How well can VALL-NUT reduce program coverage?
RQ3. How do the strategies affect the results individually?
RQ4. How is the performance of VALL-NUT compared to other anti-fuzz techniques?

B. Hindering Bug Finding (RQ1)

In this experiment, we measure the bug finding hindering capability of VALL-NUT by evaluating the number of crashes found by all fuzzers in 24 hours on both the protected binary and the original binary. Note that we did not run AFL-BB on the protected binary because VALL-NUT targets on greybox fuzzing techniques. Furthermore, we did not find crashes in Vorbis with any fuzzer during the experiment.

Figure 5 shows the average number of crashes found over time by each fuzzer in 24 hours. The lines are the average numbers of crashes; the shades are the 95% confidence intervals. The red lines are the results for AFL-QEMU; the blue lines are the results for Fairfuzz-QEMU; the black lines are the results for AFL-BB. The dotted lines are the results for the original binary; the solid lines are the results for the protected binary.

From Figure 5, we can clearly see that for each fuzzer, the number of crashes found on the protected binary is significantly less than the number of crashes found on the original binary. In fact, VALL-NUT can reduce an average of 87% crashes found by AFL-QEMU and an average of 74% crashes found by Fairfuzz-QEMU. The significance is clearly demonstrated as the lower bounds of the 95% confidence intervals of the results on the original binary are higher than the upper bounds of the 95% confidence intervals of the results on the protected binary for each fuzzer. There is only one exception where the confidence intervals of the results for Fairfuzz-QEMU on mp42aac have overlaps. Nevertheless, Fairfuzz-QEMU still finds fewer crashes on the protected binary on average. Moreover, we can also see that on exiv2, mjs and mp42aac, VALL-NUT reduces the bug finding capability of AFL-QEMU so much that it performs even worse than its blackbox counterpart.

Answer to RQ1: We can positively answer RQ1 that VALL-NUT can *significantly* hinder the bug finding capability of greybox fuzzers

C. Reducing Program Coverage (RQ2)

In this experiment, we measure the program coverage reduction capability via evaluating the number of covered edges as well as kept seeds (aka *paths* in AFL) by all fuzzers in 24 hours. Same as the experiments for hindering bug finding, we did not run AFL-BB on the protected binary.

Table II shows the detailed results of edge coverage and statistical test. For AFL-QEMU and Fairfuzz-QEMU, column *orig* shows the number of edges covered by the original

TABLE II: Edge Coverage Results (red* represents the edge reduction ratio, calculated by $(orig - anti)/orig$)

Target	AFL-QEMU					Fairfuzz-QEMU					AFL-BB			
	orig	anti	p-value	A_{12}	red*	orig	anti	p-value	A_{12}	red*	orig	p-value	A_{12}	red*
exiv2	**9506**	**7167**	**1e-4**	**1**	**25%**	**7804**	**4315**	**9e-5**	**1**	**45%**	**8776**	**9e-5**	**1**	**18%**
nm	**7551**	**1865**	**9e-5**	**1**	**75%**	**7891**	**2013**	**9e-5**	**1**	**74%**	**3432**	**9e-5**	**1**	**46%**
objdump	**5715**	**3323**	**9e-5**	**1**	**42%**	**5918**	**3427**	**9e-5**	**1**	**42%**	**4372**	**9e-5**	**1**	**24%**
mp42aac	**2588**	**2030**	**9e-5**	**1**	**22%**	**2355**	**2072**	**1e-3**	**0.9**	**12%**	2355	8e-2	1	14%
mjs	**3320**	**2791**	**9e-5**	**1**	**16%**	**3321**	**2733**	**9e-5**	**1**	**18%**	**3185**	**9e-5**	**1**	**12%**
vorbis	**2023**	**1535**	**9e-5**	**1**	**24%**	**1901**	**1683**	**2e-5**	**0.71**	**11%**	**1970**	**4e-4**	**1**	**22%**
Average	**5117.17**	**3118.5**	**9e-5**	**1**	**34%**	**4865**	**2707.17**	**2e-4**	**0.94**	**34%**	4015	1e-2	1	23%

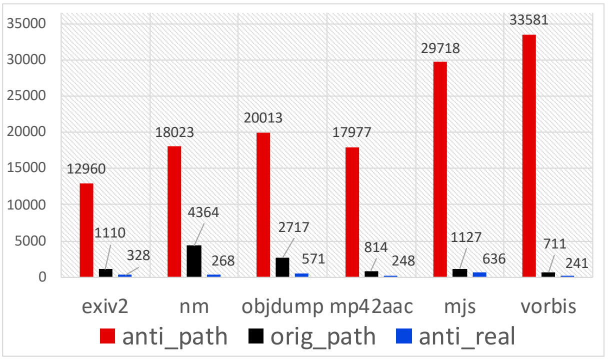


Fig. 6: Number of Paths W/-, W/O VALL-NUT

program; column *anti* shows the number of edges covered by the protected program; column *p-value* shows the Mann Whitney U-test results and a *p-value* smaller than 0.05 means the difference between the two sets of data are statistically significant; column A_{12} shows the probability that the fuzzer covers less edge on the protected binary than on the original binary; column *red* shows the percentage of the reduced edges. For AFL-BB, column *orig* shows the number of edges covered by the original program; the rest shows the statistical test results against that of AFL-QEMU on the protected program.

From Table II, we can see that all the *p-values* between the protected binary and the original binary for AFL-QEMU and Fairfuzz-QEMU are smaller than 0.05. Moreover, all the A_{12} values are higher than 0.71 – the conventional large effect size [28]. This indicates that VALL-NUT can significantly reduce the program coverage of the greybox fuzzers from the statistics perspective. In particular, by comparing the results of AFL-QEMU and AFL-BB, we can see that with the effects of VALL-NUT, AFL-QEMU finds an average 23% less edges. This means that VALL-NUT restrains the advantages of greybox fuzzing and can make it even worse than blackbox fuzzing.

Figure 6 shows the number of seeds kept by the AFL-QEMU on both protected and original binaries. The number of seeds is also known as paths in the context of AFL, because each kept seed is considered to execute a unique path. The red bar shows the average number of seeds in the queue when running on the protected binary. The black bar shows the average number of seeds in the queue when running on the original binary. The blue bar shows the average number of seeds in the queue that actually executes the unique real paths (not the paths introduced by the fuzzing obstacle code) when running on the protected binary.

By comparing the red and black bars, we can clearly see that the seed queue explosion mechanism tricks AFL-QEMU to keep a large number of seeds in queue. While the total number of seeds is high, we can see that the number of seeds exercising the real paths is small as the blue bar is always much lower than the black bar. To conclude, VALL-NUT can reduce the real path coverage of greybox fuzzers by introducing an overwhelming amount of faked paths.

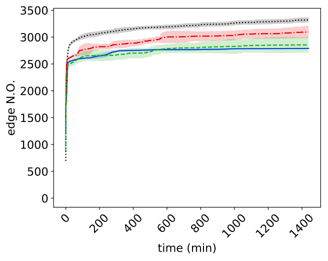
Answer to RQ2: We can positively answer RQ2 that VALL-NUT can *significantly* hinder the program coverage of greybox fuzzers and make them perform worse than blackbox fuzzers.

D. Effects of Individual Strategies (RQ3)

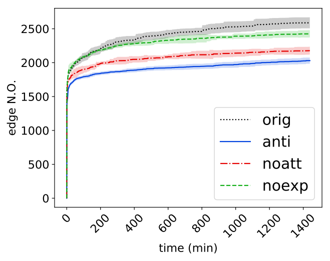
In this experiment, we evaluate the effects of individual strategies in detail with *mjs* and *mp42aac* fuzzed by AFL-QEMU. As discussed in Section IV-B, the *feedback contamination* is tightly combined with the other 2 strategies in Algo. 1, making it hard to be separated for individual evaluation. Therefore we added two sets of configurations during this evaluation. The first one is called *noatt*. As indicated by its name, we removed the code most related to *seed attenuation* (line 18 onward) from the fuzzing obstacle code in this configuration. The second one is called *noexp* and we removed the code most related to *seed explosion* by removing the first jump table. Despite the 2 newly added configurations, we still keep the results of AFL-QEMU running on the original and protected binaries for comparison.

In Figure 7 and 8, the black plots show the results of the original binary; the red plots show the results of the binary protected without *seed attenuation*; the green plots show the results of the binary protected without *seed queue explosion*; the blue plots show the results of the fully protected binary.

Figure 7 shows the number of edges covered over time by AFL-QEMU on the 4 binaries. In general, we can see that AFL-QEMU covered the least edges on the fully protected binary and covered the most edges on the original binary. The two strategies can both help to reduce the program coverage

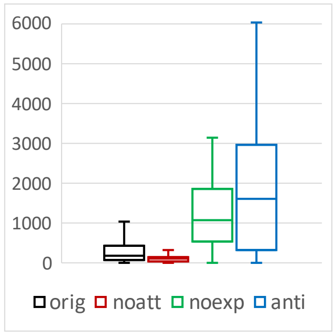


(a) mjs

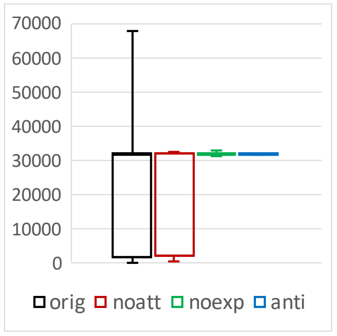


(b) mp42aac

Fig. 7: Edges Covered Over Time for Each Binary
*Lower is better.



(a) mjs



(b) mp42aac

Fig. 8: Generated Seed Size (*bytes*) for Each Binary
*Higher is better.

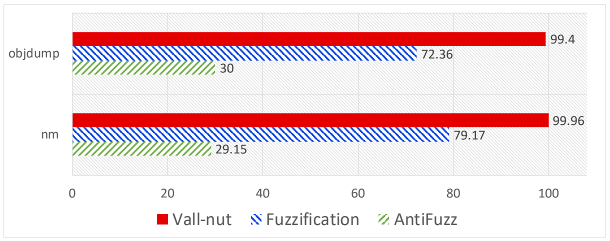


Fig. 9: Average Reduced Percentages of Crashes in VALL-NUT, Fuzzification, and AntiFuzz
*Longer is better.

individually and combining them together gives us the best result. For individual strategies, we can see that on mjs, *no-exp* performs better while on mp42aac, *no-att* performs better. This indicates that *seed attenuation* performs a more important role on mjs than on mp42aac.

To further study the reason behind this phenomenon, we plot distribution of the size of the seeds kept by the fuzzer as boxplots in Figure 8. From Figure 8, we can clearly see that the increment of seed size caused by the *seed attenuation* strategies on mjs is much more significant than on mp42aac. This is caused by the relation between the size of the initial seeds supplied to the fuzzer and the maximum value in the *input_length_steps[]* configurable variable. In section V-A, we mentioned that the configuration we used for the experiments will guide the fuzzer to eventually generate seed inputs larger than 8000 bytes. As discussed in the application scenario (Section II), this configuration is decided by the developers before distributing the program binary and the developers have no idea of the size of the initial seeds used by the attackers. The average size of the initial seeds used by mjs in the experiment is about 400 bytes, while the size of the initial seeds used by mp42aac is about 17k bytes. This means that the initial seeds of mp42aac are out-of-bound for the guidance of the *seed attenuation* strategy which further explains why the increment in average seed size on mp42aac is ignorable. As a result, the effect of *seed attenuation* strategies is much less significant on mp42aac.

Answer to RQ3: The strategies can help individually and when combined together, they can achieve the maximum performance. The significance of individual strategies varies under different circumstances.

E. Comparison with Other Anti-fuzz Techniques (RQ4)

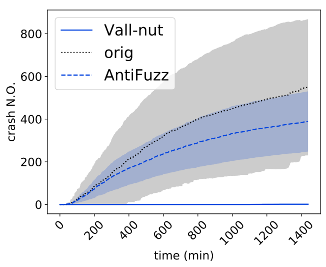
Two recent works, namely FUZZIFICATION [9] and ANTIFUZZ [10], also cover the topic of anti-fuzz. Both FUZZIFICATION and ANTIFUZZ have released their source code for public evaluation [29], [30]. We conducted additional experiments to evaluate VALL-NUT and the two related works.

For FUZZIFICATION, we met a bug during the build process of the protected program with the LLVM passes provided in the project. Later, we verified that this is the same issue as the one reported by *lawyer61* on github [31]. This error happens when the LLVM pass is adding the *SpeedBump* strategy. By the time of writing this paper, the bug was not fixed. Thus we reclaim the data in their paper for comparison.

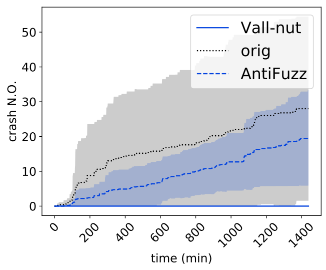
For ANTIFUZZ, the authors used a python script to generate a `.h` file containing all the functions needed for anti-fuzzing and the user will need to include the `.h` file in their project and manually insert the corresponding functions into the correct locations in the source code. Since the file containing the core logic is a `.h` file, not a `.hpp` file, the implementation of ANTIFUZZ cannot be applied to some C++ projects due to some compatibility issues such as the implicit/explicit type conversion problem. So we decided to conduct experiments on `nm` and `objdump` because they are also used by FUZZIFICATION and are written in C. Note that in our experiments, we did not enable the delaying execution feature in ANTIFUZZ because we think the 750ms sleep used in their previous experiments brings too much unfairness and VALL-NUT's effect of slowing down the execution speed of the original program is negligible (see Section VI for the discussion on execution speed delay). In addition, these experiments are repeated ten times to get statistically sound results.

Figure 9 shows the average percentages of reduced crashes found by AFL-QEMU for the program binaries protected by VALL-NUT, FUZZIFICATION and ANTIFUZZ. We can see that VALL-NUT reduces the most crashes by eliminating almost 100% of the crashes found in these 2 programs.

Figure 10 shows the number of detected crashes over time for AFL-QEMU on the original program, the program protected by ANTIFUZZ and the program protected by VALL-NUT. The line is the average value across the ten experiments and the shaded area is the 95% confidence interval. This figure shows that VALL-NUT hided significantly more crashes from AFL-QEMU

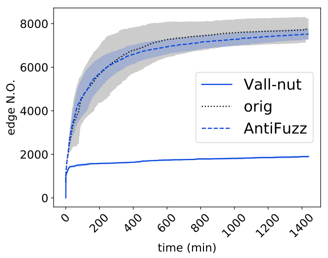


(a) nm

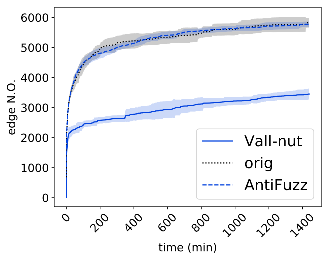


(b) objdump

Fig. 10: Number of Detected Crash Over Time
*Lower is better.



(a) nm



(b) objdump

Fig. 11: Number of Edges Covered Over Time
*Lower is better.

than ANTIFUZZ.

Figure 11 shows the number of covered edges over time for AFL-QEMU on the original program, the program protected by ANTIFUZZ and the program protected by VALL-NUT. The line is the average value across the ten experiments and the shaded area is the 95% confidence interval. This figure shows that VALL-NUT can significantly reduce the coverage achieved by AFL-QEMU while ANTIFUZZ hardly reduced the coverage.

Answer to RQ4: We can positively answer RQ4 that VALL-NUT has significant advantages comparing to existing anti-fuzz techniques in both crash and coverage reduction.

VI. DISCUSSION

Anti-VALL-NUT. As anti-greybox-fuzzing techniques advance, attackers start to study *anti*-anti-greybox-fuzzing techniques. This is an arms race between the attackers and the defenders that never ends. Moreover, whether attackers can bypass our techniques also depends on the assumptions about the skill levels of the attackers. Thus, to limit the scope of the discussion, we discuss the most straightforward attacks against VALL-NUT.

Since we use edge-level frequency profile in VALL-NUT to identify the locations for injecting the fuzzing obstacle code, a question pops up naturally: Can an attacker use edge-level frequency profiles to identify and further bypass VALL-NUT? The short answer is NO. The first thing to mention is that the attacker can only get the protected binary as discussed in our application scenario (Cf. Figure 2). The second thing is that the edge-level frequency profile is disrupted by the injected fuzzing obstacle code. This means that the attacker cannot simply use profiling techniques to identify the fuzzing obstacle code. This is further supported by the experiment results on Fairfuzz. We purposely choose Fairfuzz for evaluation because it has some mechanisms to target low-frequency (non-error handling) edges, which may lower the effectiveness of VALL-NUT. The experiment results show that such mechanism cannot prevent Fairfuzz from getting compromised as it finds 74% less crashes and covers 34% less edges when running on the protected binary for 24 hours.

In VALL-NUT, jump tables are used to carry out the strategies. Like proposed in FUZZIFICATION, we can use the `rop` gadgets to implement the jump tables to disguise them [9]. This improvement in implementation can help to increase the difficulty for the attacker to identify the fuzzing obstacle code. Moreover, as discussed in FUZZIFICATION and ANTIFUZZ, we can also apply obfuscation techniques which can further increase the effort for identifying and evading the VALL-NUT related code. To conclude, as far as the key strategies of VALL-NUT can be carried out, different implementation optimizations can be adopted and different complementary techniques can be combined to deter the attacker from applying greybox fuzzing to the protected program.

Performance overhead of VALL-NUT. As discussed in the problem scope (Cf. Section II), we try not to affect the normal functionality and execution efficiency as much as possible. Unlike [9] and [10], VALL-NUT does not apply strategies to intentionally delay the execution speed of the binary for hindering the fuzzing process. As a result, the performance overhead of VALL-NUT is very marginal. We conducted two extra experiments to evaluate the execution speed of both the protected binary and the original binary valid inputs. For the first experiment, we execute each binary 10000 times and calculate their average execution time with another set of valid inputs (not used in profiling). The data and statistical test results are in Table III. From Table III, we can see that the A_{12} results are all 0.50, meaning that there is a half and half chance that the protected binary is faster or slower. The results of the first experiment suggest that the performance overhead of VALL-NUT on normal usage is ignorable.

For the second experiment, we execute each binary 10000 times and calculate their average execution time with an invalid input which is guaranteed to execute the injected fuzzing obfuscation code. The data and statistical test results are in Table IV. From the results, we can see that the program runs faster with invalid inputs than with valid inputs. Although the average execution time of the protected program is longer than the average execution time of the original program, the difference is marginal. The A_{12} values are also around 0.50, meaning that the protected program is not more likely to execute slower than the original binary. The results of the second experiment show that the fuzzing obstacle code of VALL-NUT does not significantly slow down the execution of the original program. Note that although the slow down effect of the fuzzing obstacle is not significant, profiling and injecting the obstacle code on error handling paths are still needed because the *seed queue explosion* strategy requires to fill up the greybox fuzzer's seed pool with low quality inputs.

TABLE III: Execution Time W/-, W/O VALL-NUT on valid inputs

Target	Protected Bin	Original Bin	A_{12}
exiv2	2.09ms	2.09ms	0.50
nm	1.03ms	1.04ms	0.50
objdump	1.03ms	1.04ms	0.50
mp42aac	1.05ms	1.05ms	0.50
mjs	1.02ms	1.02ms	0.50
vorbis	2.00ms	2.02ms	0.50

TABLE IV: Execution Time W/-, W/O VALL-NUT on invalid inputs

Target	Protected Bin	Original Bin	A_{12}
exiv2	2.01ms	1.97ms	0.51
nm	1.03ms	1.00ms	0.51
objdump	1.00ms	1.00ms	0.50
mp42aac	1.00ms	1.00ms	0.50
mjs	1.00ms	1.00ms	0.50
vorbis	1.00ms	1.00ms	0.50

In addition to the experiments of running the programs independently, we also collected the execution speed of AFL-QEMU when fuzzing the protected programs and the original program. Table V shows the collected speed data. It is observed that although the speed difference between the protected program and the original program is marginal through the analysis of data from Table III and Table IV, the speed of AFL-QEMU when fuzzing the protected program is significantly slower than when fuzzing the original program. The reason of this phenomenon is that the pool of seed inputs maintained by AFL-QEMU is very different. Because of the *seed attenuation* effect of the protected programs, AFL-QEMU will keep larger seeds for the protected programs. Due to some memory operation pitfalls, the fuzzer needs a longer time to handle larger seed inputs [32] and thus executes slower. To conclude, VALL-NUT can slow down the execution speed of the fuzzer without hindering the performance of the protected program.

VII. RELATED WORK

Anti-fuzzing Techniques. There are a few works on anti-fuzzing. Whitehouse *et al.* [7] proposed a number of anti-fuzzing strategies to resist vulnerability detection, such as fake crashes, performance degradation. David *et al.* [8] studied the feedback mechanisms that are commonly used by modern fuzzers, and then masked the signals emitted from the abnormal code, leading to wrong feedback. As such, the fuzzers can be misled and degraded with wrong information. Kang *et al.* apply fake code injection to prevent AFL in QEMU mode from finding a specific crash site. VALL-NUT is much more general in that it accounts for different mainstream fuzzers and does not target *specific paths that may be vulnerable*. Contemporary with our work, Jung *et al.* [9] and Güler *et al.* [10] find some weaknesses of modern fuzzers and propose countermeasures to nullify or degrade the fuzzing advantages. Differently, we conduct a systematic study on the inspiring mechanism of coverage feedback in modern fuzzers, and identify the fundamental links that power the outperformance. The strategies proposed in this paper can entirely comprise these links and achieve a generic defense against greybox fuzzers. For example, our approach outperforms FUZZIFICATION in terms of reducing detected crashes as shown in Figure 9.

TABLE V: Fuzzing Speed W/-, W/O VALL-NUT

Target	Protected Bin	Original Bin	A_{12}
exiv2	4 exec/s	27 exec/s	1.0
nm	70 exec/s	276 exec/s	1.0
objdump	57 exec/s	255 exec/s	1.0
mp42aac	371 exec/s	627 exec/s	1.0
mjs	78 exec/s	336 exec/s	1.0
vorbis	53 exec/s	118 exec/s	1.0

Greybox Fuzzing. VALL-NUT highly inspires from the recent advances in greybox fuzzing techniques, which majorly root in the improvements in *seed evaluator*, *seed mutator* and *feedback collector* (Cf. Figure 1). Some fuzzing techniques modify the preference on the seeds for different fuzzing purposes. For example, AFLFast [11] evaluates the seeds based on a Markov chain model to improve coverage; AFLGo [33] and Hawkeye [34] adjust seeds' priorities based on their distances to the targets. Other techniques focus on improving seed mutators to increase mutation effectiveness, either by reducing ineffective mutators for specific inputs [14], [35], or applying structure-aware mutations on target programs [36]–[39]. Several other works [15], [40]–[44] collect more feedback to distinguish different execution states, which in turn helps to improve the seed evaluation and mutation. To impede these techniques, VALL-NUT decreases their effectiveness on the opposite site: VALL-NUT confuses the seed evaluator to value those seeds trapped into our injected code obstacles; VALL-NUT applies *seed attenuation* to distort the seed mutator to generate more large seeds that decrease the overall fuzzing efficiency; VALL-NUT contaminates the feedback with saturated records and non-deterministic execution behaviors. As a result, VALL-NUT can hinder different fuzzing techniques systematically and effectively.

Anti-analysis Techniques. Making code uninterpreted and unanalyzed is another efficient way to prevent vulnerability detection. Collberg *et al.* [45] present a number of obfuscating transformations to prevent the analysis of intellectual properties. Obfuscation proved to be effective in collapsing widely used static analyzer and symbolic executor [46]–[49]. Others approaches suggest injecting chaff code or even bugs to fight against bug analysis tools and attackers' efforts. For example, Hu *et al.* [50] propose to insert a large number of non-exploitable bugs to increase the cost of attacking. On the other hand, software diversity is proposed to introduce uncertainties into the target program and restrict attackers from inferring the implementation details based on either static or dynamic analysis results [51]. For example, the randomization [52]–[54] towards binary can destruct the return values during execution and thereby complicate the utilization. Although the above techniques can counteract the greybox fuzzers armed with static analysis, they cannot stop these fuzzers from getting sufficient runtime feedback for vulnerability inference. VALL-NUT concentrates on attacking the feedback collection and utilization. Through designed strategies, VALL-NUT impedes or pollutes the feedback to downgrade greybox fuzzers' power of vulnerability detection.

ACKNOWLEDGMENT

We sincerely thank the anonymous reviewers who have reviewed this paper for their precious comments. Guozhu Meng was partly supported by the National Key R&D Porgrammes of China (No. 2019AAA0104301), National Natural Science Foundation of China (No. 61902395) and CCF-Tencent Open Fund. This research was supported by the National Research Foundation through its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) project under the National Cybersecurity R&D (NCR) Grant award no. NRF2018NCR-NSOE003-0001. Jun Xu was partly supported by a gift grant from Cisco.

VIII. CONCLUSION

In this paper, we present a systematic study of greybox fuzzing on its strengths and propose counteractive strategies accordingly. From the standpoint of defenders, we develop three novel strategies–seed queue explosion, seed attenuation, and feedback contamination to restrain the advantages of greybox fuzzing. Our evaluation shows that VALL-NUT can effectively downgrade the performance of greybox fuzzing, making it even worse than blackbox fuzzing.

REFERENCES

[1] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, no. 12, 1990. [Online]. Available: http://doi.acm.org/10.1145/96267.96279

[2] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, New York, NY, USA, 2013. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516736

[3] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *ACM Sigplan Notices*, no. 6. ACM, 2008.

[4] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *ICSE*, 2009.

[5] V. Pham, M. Böhme, and A. Roychoudhury, "Model-based whitebox fuzzing for program binaries," in *ASE*. ACM, 2016.

[6] "American fuzzy lop," http://lcamtuf.coredump.cx/afl/, accessed: 2018-04-01.

[7] O. Whitehouse, "Introduction to Anti-Fuzzing: A Defence in Depth Aid," https://www.nccgroup.trust/sg/about-us/newsroom-and-events/blogs/2014/january/introduction-to-anti-fuzzing-a-defence-in-depth-aid/, (Accessed on 02/02/2018).

[8] D. G. E. Edholm, "Escaping the fuzz: Evaluating fuzzing techniques and fooling them with anti-fuzzing," 2016.

[9] J. Jung, H. Hu, D. Solodukhin, D. Pagan, K. H. Lee, and T. Kim, "Fuzzification: Anti-fuzzing techniques," in *the 28th USENIX Security Symposium (USENIX 2019)*. USENIX, 2019.

[10] E. Güler, C. Aschermann, A. Abbasi, and T. Holz, "Antifuzz: Impeding fuzzing audits of binary executables," in *the 28th USENIX Security Symposium (USENIX 2019)*. USENIX, 2019.

[11] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016.

[12] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing," in *NDSS*, 2017. [Online]. Available: https://www.vusec.net/download/?t=papers/vuzzer_ndss17.pdf

[13] M. Böhme, V. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, 2016. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978428

[14] C. Lemieux and K. Sen, "Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage," *CoRR*, 2017. [Online]. Available: http://arxiv.org/abs/1709.07101

[15] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *Proceedings of the 39th IEEE Symposium on Security and Privacy*, 2018.

[16] "Technical "whitepaper" for afl-fuzz," http://lcamtuf.coredump.cx/afl/technical_details.txt, accessed: 2018-04-01.

[17] A. Helin. (2018) radamsa - a general-purpose fuzzer. [Online]. Available: https://gitlab.com/akihe/radamsa

[18] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673

[19] Anonymous. (2019) Redis, an open source, in-memory data structure store database/cache/message broker. [Online]. Available: https://redis.io/

[20] C. Lv, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "Mopt: Optimized mutation scheduling for fuzzers," in *the 28th USENIX Security Symposium (USENIX 2019)*. USENIX, 2019.

[21] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang, "Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery," in *Proceedings of the 40th IEEE Symposiums on Security and Privacy*, 2019.

[22] B. Mathis, R. Gopinath, M. Mera, A. Kampmann, M. Höschele, and A. Zeller, "Parser-directed fuzzing," in *40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*, 2019. [Online]. Available: https://publications.cispa.saarland/2823/

[23] G. Inc., "Google fuzzer testing suite," https://github.com/google/fuzzer-test-suite, 2019.

[24] W. You, X. Liu, S. Ma, D. Perry, X. Zhang, and B. Liang, "Slf: Fuzzing without valid seed inputs," in *Proceedings of the 41st ACM/IEEE International Conference on Software Engineering, ICSE*, 2019.

[25] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018.

[26] R. L. Ott and M. T. Longnecker, *An introduction to statistical methods and data analysis*, 2015.

[27] A. Arcuri and L. Briand, in *Software Engineering, 2011 33rd International Conference on*. IEEE, 2011.

[28] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, no. 2, 2000.

[29] (2019) Antifuzz: Impeding fuzzing audits of binary executables. [Online]. Available: https://github.com/RUB-SysSec/antifuzz

[30] (2019) About fuzzification. [Online]. Available: https://github.com/sslab-gatech/fuzzification

[31] (2019) About fuzzification. [Online]. Available: https://github.com/sslab-gatech/fuzzification/issues/3

[32] (2019) Some thoughts on fuzzing. [Online]. Available: https://gamozolabs.github.io/2020/08/11/some_fuzzing_thoughts.html

[33] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM Press, 2017, pp. 2329–2344.

[34] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzer," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: ACM, 2018, pp. 2095–2108. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243849

[35] C. Lv, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "Mopt: Optimize mutation scheduling for fuzzers," p. N.A.

[36] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proceedings of the 21st USENIX Conference on Security Symposium*, ser.

Security'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 38–38. [Online]. Available: http://dl.acm.org/citation.cfm?id=2362793.2362831

[37] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," ser. SP '17, May 2017, pp. 579–594.

[38] V. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *CoRR*, vol. abs/1811.09447, 2018. [Online]. Available: http://arxiv.org/abs/1811.09447

[39] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," *CoRR*, vol. abs/1812.01197, 2018. [Online]. Available: http://arxiv.org/abs/1812.01197

[40] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: Program-state based binary fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, New York, NY, USA, 2017. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106295

[41] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," *CoRR*, 2018. [Online]. Available: https://arxiv.org/abs/1803.01307v2

[42] C. Wen, H. Wang, Y. Li, S. Qin, Y. Liu, Z. Xu, H. Chen, X. Xie, G. Pu, and T. Liu, "Memlock: Memory usage guided fuzzing," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 765777. [Online]. Available: https://doi.org/10.1145/3377811.3380396

[43] H. Wang, X. Xie, Y. Li, C. Wen, Y. Li, Y. Liu, S. Qin, H. Chen, and Y. Sui, "Typestate-guided fuzzer for discovering use-after-free vulnerabilities," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 9991010. [Online]. Available: https://doi.org/10.1145/3377811.3380386

[44] H. Chen, S. Guo, Y. Xue, Y. Sui, C. Zhang, Y. Li, H. Wang, and Y. Liu, "MUZZ: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2325–2342. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/chen-hongxu

[45] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," The University of Auckland, Tech. Rep. 148, 1997.

[46] A. Balakrishnan and C. Schulze, "Code obfuscation literature survey," 04 2015.

[47] "Diversification and obfuscation techniques for software security: A systematic literature review," *Information and Software Technology*, vol. 104, pp. 72 – 93, 2018.

[48] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. R. Weippl, "Protecting software through obfuscation: Can it keep pace with progress in code analysis?" *ACM Computing Surveys*, vol. 49, no. 1, pp. 4:1–4:37, 2016.

[49] V. Balachandran, Sufatrio, D. J. J. Tan, and V. L. L. Thing, "Control flow obfuscation for android applications," *Computers & Security*, vol. 61, pp. 72–93, 2016. [Online]. Available: https://doi.org/10.1016/j.cose.2016.05.003

[50] Z. Hu, Y. Hu, and B. Dolan-Gavitt, "Chaff bugs: Deterring attackers by making software buggier," *CoRR*, vol. abs/1808.00659, 2018. [Online]. Available: http://arxiv.org/abs/1808.00659

[51] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*. IEEE Computer Society, 2014, pp. 276–291. [Online]. Available: https://doi.org/10.1109/SP.2014.25

[52] E. Shioji, Y. Kawakoya, M. Iwamura, and T. Hariu, "Code shredding: byte-granular randomization of program layout for detecting code-reuse attacks," in *28th Annual Computer Security Applications Conference, ACSAC 2012, Orlando, FL, USA, 3-7 December 2012*, 2012, pp. 309–318.

[53] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, 2003, pp. 272–280.

[54] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "ILR: where'd my gadgets go?" in *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, 2012, pp. 571–585.