Bear: A Framework for Understanding Application Sensitivity to OS (Mis)Behavior

Ruimin Sun*, Andrew Lee*, Aokun Chen*, Donald E. Porter†, Matt Bishop‡, and Daniela Oliveira*

*University of Florida, Email: gracesrm,andrewlee,aokunchen1990@ufl.edu, daniela@ece.ufl.edu
†University of North Carolina at Chapel Hill, Email: porter@cs.unc.edu
‡University of California at Davis, Email: mabishop@ucdavis.edu

***Abstract*—Applications are generally written assuming a predictable and well-behaved OS. In practice, they experience unpredictable misbehavior at the OS level and across OSes: different OSes can handle network events differently, APIs can behave differently across OSes, and OSes may be compromised or buggy. This unpredictability is challenging because its sources typically manifest during deployment and are hard to reproduce. This paper introduces Bear, a framework for statistical analysis of application sensitivity to OS unpredictability that can help developers build more resilient software, discover challenging bugs and identify the scenarios that most need validation. Bear analyzes a program with a set of perturbation strategies on a set of commonly used system calls in order to discover the most sensitive system calls for each application, the most impactful strategies, and how they predict abnormal program outcome. We evaluated Bear with 113 CPU and IO-bound programs, and our results show that null memory dereferencing and erroneous buffer operations are the most impactful strategies for predicting abnormal program execution and that their impacts increase ten-fold with workload increase (e.g. number of network requests from 10 to 1000). Generic system calls are more sensitive than specialized system calls—for example, `write` and `sendto` can both be used to send data through a socket, but the sensitivity of `write` is twice that of `sendto`. System calls with an array parameter (e.g. `read`) are more sensitive to perturbations than those having a struct parameter with a buffer (e.g `readv`). Moreover, the fewer parameters a system call has, the more sensitive it is.**

I. Introduction

Applications are generally written with the assumption that the OSes on all deployed systems will behave predictably and identically to the testing environment. Recent research [1]–[3], however, has shown that unpredictability and misbehavior at the OS level are more common than once thought. Unpredictability in OS behavior can cause bugs and security vulnerabilities in applications. These bugs or vulnerabilities can arise from: (i) different ways OSes handle network events and protocols [1], and (ii) subtle and undocumented differences in the behavior of common APIs across different platforms [2] and from OS changes over time. Further, the OS can be buggy or malicious, which breaks the predictability assumption. For example, in Iago attacks [3], a malicious kernel induces a protected process to act against its interests by manipulating system call return values. These application bugs and vulnerabilities are hard to reproduce in a testing environment, and thus are difficult to detect and prevent. Although recent technology trends such as Intel SGX are attempting to introduce mutual distrust between the OS and its applications [4]–[7], such as by protecting the application's memory from the OS, a significant burden still falls to the application developer to reason about the *semantic impact* of return values from the OS on their applications. In reality, developers are simply not equipped to write robust applications in the face of unpredictable or even adversarial OSes.

Developers need techniques to understand the impact of OS unpredictability and misbehavior on program execution. With a better understanding of this sensitivity, application developers can (i) discover bugs that would only appear after program deployment and would be hard to reproduce, (ii) efficiently target end-to-end checks as well as time-consuming testing and verification procedures, and (iii) design and implement applications that can withstand an OS's malicious or buggy misbehavior [3]. This understanding can also improve OS design, allowing designers to introduce diversity into software without affecting application execution, thereby alleviating the security shortcomings of today's software monoculture. OS developers currently fear any variability short of bug-for-bug compatibility, lest the OS harm benign programs; with an understanding of which behaviors programs were sensitive or insensitive to, OSes could diversify behavior and implementation.

This paper introduces Bear, a Linux-based framework for fine-grained statistical analysis of application sensitivity to OS unpredictability. Bear statistically analyzes a program using a set of unpredictability strategies on a set of commonly used system calls in order to discover the most sensitive system calls for each application, the most impactful strategies, and how they predict abnormal program outcome (crash, segmentation fault, etc.). Rigorously understanding application sensitivity to OS misbehavior can help developers discover challenging bugs and edge cases, as well as identify the scenarios that most need end-to-end checks, targeted testing and verification procedures. Bear performs a correlation and regression analysis on the program outcome after the program is run for a number of times and for every selected combination of system call, perturbation strategy, and perturbation threshold. The correlation analysis informs whether or not, for a particular program, there is a relationship between a program execution outcome and a system call, strategy, or threshold. If there is a correlation, regression analysis predicts the likelihood of an abnormal program outcome for different system calls,

strategies and thresholds. This analysis allows a developer to have insights on the following questions: *(i) which system calls are the most sensitive to OS unpredictability and by what degree? (ii) which strategies cause the most impact in program execution and by what degree? and (iii) do program type and execution workloads affect the strategy impact or system call sensitivity?*

Our results showed that for CPU-bound applications, the four most sensitive system calls are `mmap`, `write`, `munmap` and `rt_sigprocmask`. While for I/O-bound applications, the four most sensitive system calls are `mremap`, `write`, `read` and `munmap`. For all applications, null dereferencing and buffer overflow are the two most severe events in predicting an abnormal program outcome. For CPU-specific applications, signal-related errors are also very impactful. On the other hand, network-related system calls such as `sendto` and `recvfrom` have not shown high significance on abnormal execution, which may be the result of robust end-to-end checkings and retry mechanisms over network layer. The two least impactful strategies are naked notify in method and privilege degradation. The rest of the strategies are of similar impact.

Generic system calls are more sensitive than specialized system calls, for example, `write` and `sendto` can both be used to send data through a socket, but the sensitivity of `write` is twice as of `sendto`. System calls with an array parameter (e.g. `read`) are more sensitive to perturbations than those having a struct parameter with a buffer (e.g `readv`). Moreover, the fewer parameters a system call has, the more sensitive it is.

When the program workload is heavy, the impact of buffer overflows and wrong parameter types in file-related system calls on abnormal program execution sharply increased compared with the medium workload and far exceeded the impact of null dereferencing errors. The perturbation strategy impact and the system call sensitivity to perturbation were not affected when we changed the program workloads from light to medium.

Programmers should be very careful when handling system calls with a buffer parameter, considering how commonly used `read` and `write` system calls are, especially when application workload is heavy. Null dereferencing is a severe problem as well and almost the hardest to debug when a segmentation fault occurs. Therefore, failure-oblivious computing can be a promising way for saving developers and testers from memory bugs. In a resource-constrained development environment (time, human resources, performance constraints), developers should prioritize testing and the inclusion of end-to-end checks for system calls that are specialized, having few parameters and having a struct parameter that includes a vector.

This work makes the following contributions:

- It represents the first statistical analysis of the impact of OS unpredictability on program execution, to the best of our knowledge.
- It provides a framework for developers to find weak spots in their code that are sensitive to OS misbehavior, facilitating more resilient and secure software.
- It can provide a new way to introduce diversity into OS execution (thus improving security) by leveraging combinations of system calls and strategies that cause unpredictability in system execution, but at the same time do not harm the software.

The paper is organized as follows. Section II describes Bear's design, while Section III details its implementation. Section IV gives an overview of the statistical methods used in this work. Section V summarizes our experimental evaluation. Section VI describes related work in the area of fuzzy testing, fault-tolerance, and program diversity, and Section VII concludes the paper.

II. Design

In this section we discuss Bear's design, whose goal is to allow a developer to analyze a target program against various types of OS unpredictability. We chose to design and implement Bear in Linux because of the Linux's open source. This design can be ported to other modern OSes. Bear is comprised of several modules residing both in user space and in the OS kernel. In summary, a testing harness component (*Manager*) loads a module (*Perturbation Module*) in the kernel that implements a set of perturbation strategies on a set of system calls (*Perturbation Set*). The *Manager* also runs the target program and invokes a statistical module for analysis.

Figure 1 illustrates Bear's general architecture. More specifically, Bear runs the application inside of a testing harness, called the *Manager*. The manager is responsible for receiving user input and invoking all other components. The user (typically an application developer) inputs the target program, a test case, the set of system calls to be perturbed, the set of perturbation strategies to be used, a perturbation threshold, and a timeout for hung programs.

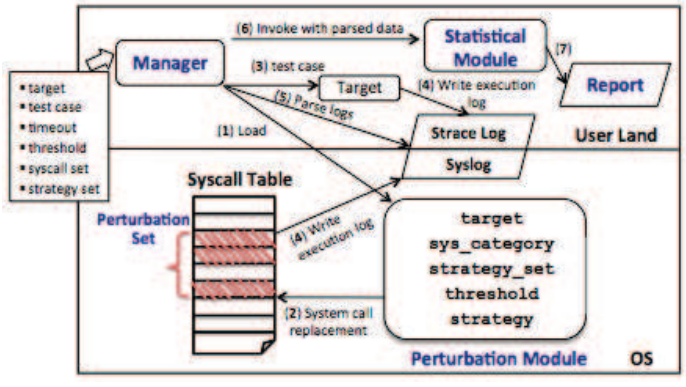

Fig. 1: Bear's architecture.

The user can select to perturb all system calls or can specify a system call category for perturbation—for example, file management, signal control, process scheduling, and network

communication. She can also specify that all perturbation strategies should be used during the analysis, or she can select a particular strategy. For example, consider a developer running Bear and passing Chrome as the target program, "All" as the set of system calls, and 10% as the perturbation threshold. Bear will perform a series of program runs with the test case, where in each run it applies one of the strategies to a system call in the selected set with 10% probability. In the Chrome example, during the analysis of `sys_read` approximately 10% of its invocations will be perturbed with a certain strategy. Each run represents a combination $\{sys_read, strategy_i\}$, where i represents each strategy that can be applied on `sys_read`.

Bear runs each combination $\{sys_read, strategy\}$ a number of times for statistical significance. The next component in Bear's architecture is the *Perturbation Module* residing at the OS level. This module is loaded by the Manager (*Step 1* in Figure 1) and is responsible for implementing all perturbation strategies. Table I lists part of the system calls that can be perturbed (the *Perturbation Set*), their categories, and the perturbation strategies that can be applied to them (strategies are discussed further in this section). During initialization the *Perturbation Module* replaces the original versions of the system calls in the perturbation set with new versions implementing the perturbation strategies (*Step 2*).

The target program is then run by the *Manager* with `strace`. During the program runs (*Step 3*), the modified versions of the perturbed system calls and `strace` will record relevant information about the run, respectively in the system logger and in the `strace` logger (*Step 4*). After the program runs finish, the *Manager* captures these logs (*Step 5*) and formats them for input to the *Statistical Module* (*Step 6*).

The *Statistical Module* then performs a correlation analysis to discover (i) whether there is a relationship or a correlation between a system call, strategy, and the program outcome (normal or abnormal), and (ii) the strength of this relationship. Next, the *Statistical Module* performs a regression analysis to predict the likelihood of an abnormal program outcome (an erroneous exit) for each system call and strategy analyzed. Finally, it writes the results of the analysis in a report for the developer (*Step 7*). The statistical tests used in this analysis are detailed in Section IV.

A. Perturbation Strategies

The *Perturbation Module* applies error patterns to the *Perturbation Set*. For this work, we chose common OS system calls from various categories: memory management, signal control, file operation, network communication and process scheduling.

The perturbation set (Table I) targets major system misbehavior that will cause common software bugs and security flaws [9], such as memory leak, synchronization error, values outside domain, buffer overflow, etc. For this work, we introduced a set of perturbation strategies to cover the following scenarios of OS misbehavior:

- **Fail to deallocate a system call (failMem)**: This misbehavior strategy simulates a memory leak when a program's allocated memory is not freed subsequently, which can cause the program to substantially increase its memory usage and crash. We implement this strategy by returning -1 when `sys_unmap()` is invoked.
- **Empty buffer in memory system call (nullMem)**: This strategy simulates a missing initialization, which can lead to a NULL reference error. This strategy is implemented by changing the buffer parameter to `null` for memory-related system calls, such as `sys_mmap()`.
- **Fail of lock related system call (failLock)**: This strategy simulates synchronization errors (deadlocks, race conditions and live lock) in the control of the execution of multi-threads programs with shared data. We implement this strategy by returning -1 when lock-related system calls are invoked, such as `sys_mlock()`.
- **Failure to signal control system call (failSig)**: This strategy simulates signal delivery error and returns -1 when a signal control system call is invoked, such as `sys_rt_sigaction`.
- **Different data type to parameter (diffType)**: This strategy simulates an incorrect input passed, such as passing an integer parameter, where the expected type is a character. We implement this strategy by changing `char *` parameter to `int` when a system call such as `sys_write()` is invoked.
- **Injection of bytes to system call with buffer (bufOf)**: This strategy simulates a buffer overflow error and is implemented by injecting random bytes to a buffer parameter in a network or file-related system call such as `sys_write()` or `sys_sendto()`.
- **Failure to access system call (failAcc)**: This strategy simulates the failure of checking access for global volatile objects that are going to be shared between several threads, such as a library. We implement this strategy by returning -1 for the `sys_access()` system call.
- **Reduction of buffer size/length parameter (redLen)**: This strategy simulates the scenario of an incorrect return value check that can cause byte loss, if only part of a buffer passed as a system call parameter is read or written. We implement the strategy by reducing the buffer length for network and file-related system calls.
- **Fail to notify system call (failNoti)**: This strategy simulates a failed call of `sys_mq_notify()`, which can cause shared object states to be modified without knowledge. The strategy is implemented by returning -1 to `sys_mq_notify()`.
- **Fail to change user id (chUid)**: This strategy simulates a failure to change privileges, such as failing to change the user id to root when `sys_setuid` is invoked.

III. Implementation Details

Bear's *Manager* executes at the user-level and automates analysis by (i) loading the *Perturbation module* with the input parameters passed by the developer; (ii) executing the

Category	System Call Example	Strategies	Common Related Bugs
Memory Management	sys_unmap	Fail to deallocate system call (failMem)	Memory leak
	sys_mmap	Empty buffer in memory system call (nullMem)	Null dereferencing
Signal Control	sys_mlock	Failure to lock related system call (failLock)	Synchronization error
	sys_kill	Failure to signal control system call (failSig)	Signal delivery error
File Operation	sys_read	Different data type to buffer parameter (diffType)	Value outside domain
	sys_write	Injection of bytes to system call with a buffer(bufOf)	Buffer overflow
Network Communication	sys_access	Failure to access system call (failAcc)	Dealing with volatile objects
	sys_sendto	Reduction of buffer size/length parameter (redLen)	Buffer return not checked
Process Scheduling	sys_mq_notify	Fail to notify system call (failNoti)	Naked notify in method
	sys_setuid	Fail to change user id (chUid)	Privilege degradation

TABLE I: Applicable strategies and examples of system calls in the perturbation set. The full set of system calls can be found in [8]. In later analysis we use `syscall` rather than `sys_syscall` for short.

target program for each combination of system call, strategy, and threshold enough times for statistically significant results; (iii) capturing the results of each run from the `syslog` and `strace` logs; (iv) parsing the results of each run into a format required by the open-source statistical module R [10]; and (v) invoking R for analysis. The user inputs into *Manager* the target program pathname, the category of system calls to be perturbed (all, file, process, signal, or network), a perturbation threshold, a test case, and a timeout, in case a process hangs. Figure 3 illustrates the format of Bear's input and output.

Upon initialization, the *Perturbation module* replaces the system call table pointers for the system calls in the perturbation set with new versions implementing the strategies. [1] In newer versions of Linux, the system call table is read-only; we remap the table read-write. The *Perturbation module* introduces four new variables: (i) `target`, the pathname for the target program being analyzed; (ii) `sys_category`, the category of system calls being analyzed (all, file, net, process, signal, or mem); (iii) `strategy_set`, the strategy(ies) to be used in the analysis (see Section II for details); and (iv) threshold, a number in $[0, 1]$, which represents the probability of perturbation for each system call that is part of the analysis.

Algorithm 1 shows how the OS applies perturbation strategies to `sys_write`. The system call performs three checks that all must be true before a strategy is applied: (i) the current program must be `target`, (ii) the current system call to be analyzed is `sys_write`, and (iii) a randomly selected number in $[0, 1]$ should be smaller than the threshold. If all conditions are true the selected strategy is applied on the system call.

The *Manager* checks every `check_frequency` (also a configurable parameter) for whether the target program is still running by searching the program in the list of running programs. If the program exits normally, it will invoke system call `exit_group` with exit code 0. Otherwise, if the exit code is non-zero, the program ended abnormally. The exit code is recorded in the `strace` log.

Algorithm 1: Perturbing `sys_write()`

```
Function long my_sys_write(fd, buf, size)
    if (current == target and syscall == sys_write and
    (random(0.0, 1.0) < threshold)) then
        switch strategy do
            case diffType
                /* Pass different data type  */
                intNum = getRandomInt();
                return orig_sys_write(fd, intNum, size);
            end
            case redLen
                /* Reduce buffer length      */
                newsize = random(0, size);
                orig_sys_write(fd, buf, newsize);
                return size;
            end
            case bufOf
                /* Inject buffer bytes       */
                newsize = random(0, 4);
                append newsize bytes in buf ;
                orig_sys_write(fd, buf, newsize);
                return size;
            end
        endsw
    else
        return orig_sys_write(fd, buf, size);
    end
```

IV. Statistical Methods

This section explains the statistical methods underlying Bear. In our study, the independent variables are system call, perturbation strategy, program type (CPU-bound or I/O-bound), and perturbation threshold. The dependent variable is the program outcome, which can be normal or abnormal. The user of Bear manipulates one or more of these independent variables and measures the change in the dependent variable.

Our statistical analysis uses standard hypothesis testing [11]. In our context, the null hypothesis (H_0) is that the execution of a particular system call or a perturbation strategy is unassociated with the program outcome, i.e., that misbehavior of the system call will not be associated with nor will it predict abnormal program outcome. A statistical test produces a *p*-value based on the results under analysis. The *p*-value is the probability of the test making a **Type I error** [11]: *rejecting the null-hypothesis when it is actually true*. When the *p*-value is very low (0.05 or 0.01), the experimenter will only observe this error 5% or 1% of the time.

[1] We decided not to implement the strategies with `strace` because it has limitations on changing the parameters.

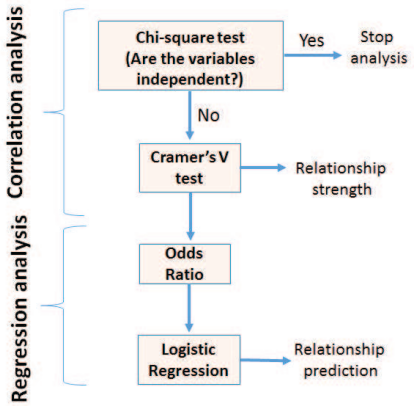

Fig. 2: Statistical tests used in this study.

Before applying the appropriate statistical test,[2], the experimenter selects an appropriate *significance level* called α, which is also a very low probability (0.10, 0.05, or 0.01). α is the experimenter's threshold for statistical significance: the test is statistically significant if the produced p-value $\leq \alpha$. Figure 2 summarizes the statistical analyses used in Bear's evaluation. We performed a *correlation analysis*, to discover and measure the strength of the association between the variables, and a *regression analysis* to discover the cause and effect of the variables' relationship. The next subsections detail each one of these tests.

A. Correlation Analysis

Correlation analysis is a statistical technique used to measure and describe the relationship between two variables. The chi-squared test [11] is a test of independence and is used for correlation of non-numerical data, such as those we have in this study: system calls (e.g., `sys_read`), strategy types (e.g., *failMem*), and program outcome (normal or abnormal). This test can determine whether there is a relationship between two variables.

The chi-squared test uses the frequency data from a sample to evaluate the relationship between two variables (e.g., perturbed system call and program outcome). Each individual in the sample (e.g. program run) is classified on both of the two variables (e.g. which system call was perturbed and program outcome), creating a two-dimensional frequency-distribution matrix: the chi-squared matrix [11].

In our case we have two matrices: one for system call and program outcome, and another for strategy and program outcome. The observed frequency f_o is the number of program runs that are classified in a particular category. The next step is to find the expected frequencies (f_e values) for the test. The expected frequencies define an ideal hypothetical distribution in agreement with the null hypothesis and are predicted from the proportions in the null hypothesis and the sample size (n).

[2]The type of statistical test depends on the goals of the experimenter, the type of the study design—single group, between groups, repeated measures, *etc.*—and the type of data—continuous, discrete, categorical, ordinal, *etc.*

The value of f_e in each cell is obtained as $f_e = (f_c \times f_r)/n$, where f_c is the frequency total for the column (column total), f_r is the frequency total for the row (row total), and n is the sample size [11].

After expected frequencies are obtained, we compute a chi-squared statistic (χ^2) to determine how well the data (observed frequencies) fit the null hypothesis (expected frequencies) as $\chi^2 = \sum (f_o - f_e)^2/f_e$. The formula measures the discrepancy between the data (f_o values) and the hypothesis (f_e values); the goal is to see how close to the expected values our observed values are. The larger the discrepancy, the larger the value for χ^2 is and the stronger the evidence is against the null hypothesis.

We can think of the chi-squared test as a binary test: it only tells us whether two variables are independent or not. As we were also interested in measuring the strength of the variables' correlation, we continued the analysis with the Cramer's V test [11]. This test builds upon the chi-squared test (it uses the matrices generated by chi-squared) and measures the strength of the variables' correlation. Cramer's V produces a value between $[0, 1]$ that measures the strength of the correlation—0.00 means no relationship and 1.00 means a perfect relationship.

B. Regression Analysis

Bear uses regression analysis to predict the relationships between perturbed system calls and program outcome, and between strategies and program outcome. While correlation analysis measures the strength of the relationship between these variables, regression analysis predicts the $odds$, defined as $p/(1-p)$, where p is the probability.

Logistic regression is a type of regression model used when the dependent variable is categorical, which is the case in our analysis. Logistic regression uses the concept of the $odds\ ratio$, which represents the constant effect of a predictor X (independent variable), on the likelihood that one outcome will occur (dependent variable) [12]. With odds ratio we can predict the likelihood of a system call or strategy causing an abnormal program outcome. For example, consider that we define system call A as reference system call and find that the odds ratio of system call B to cause an abnormal program outcome is 4. This means that system call B is 4 times or 300% more likely to cause an abnormal program outcome than A. The experimenter usually selects the most intuitive category as the reference although the choice of reference system call does not change the final prediction.

Logistic regression defines the dependent variable, Y (*program outcome*), as the logarithm of the odds and as a linear function of explanatory variables. The explanatory variables, X_i, are the logged independent variables (perturbed system call and type of strategy). The coefficient, β_i, is the independent effect of independent variable X_i on the dependent variable Y. Together, logistic regression is in the form:

$$Y = \log\left(odds(p/(1-p))\right) = \beta_0 + \beta_1 x_1 + ... + \beta_n x_n$$

Regression techniques try to find the best-fitting straight line for the data. When a simple linear regression model is fitted to logged variables, the exponentiated slope coefficient represents the predicted percent change in the dependent variable per percent change in the independent variable, regardless of their current levels [13]. For categorical variables like those used in our study, we use dummy variables of the form: $x_i = 1$ if $X_i = read$, and $x_i = 0$, otherwise.

To sum up, in our analysis we use standard hypothesis testing to understand the relationship between perturbed system calls, perturbation strategies, and program outcomes. Our analysis uses correlation analysis to discover whether these variables are indeed dependent and what the strength of their relationships are. Then our analysis employs logistic regression to predict the effects of these variables on an abnormal program outcome.

V. Evaluation

This section describes the experiments we conducted to validate Bear. We analyzed 113 applications including applications from GNU Core Utilities [14], SPEC CPU2006 [15] and Phoronix Test Suites [16]. We selected Core Utilities because their applications provide the most commonly used functions in Linux. SPEC and Phoronix benchmarks allowed our evaluation to test different workloads and provided sufficient off-the-shelf test cases. Bear framework doesn't rely on source code for white-box testing, so test coverage doesn't make a difference to our result. For the CoreUtils applications, we selected test case as their commonly used functions. The studied applications fall into four categories—processor (70 applications), network (11 applications), disk (27 applications) and encoding (5 applications).

Our evaluation lasted for 200 hours with approximately 100,000 runs in total. Each run is a combination of the following tuple: $\{program, syscall, strategy, threshold\}$. The evaluation covered every system call and strategy in Table I and every threshold in $\{10\%, 50\%, 90\%\}$. Each combination ran five times and the number of runs per combination (five) was determined empirically based on the requirements of the chi-squared tests [17]: the chi-square matrix's expected counts should be at least five counts in each cell.

We installed the Bear framework on a virtual machine with 1GB RAM, 40GB Hard Disk, x86_64 architecture, and single processor running Ubuntu 12.04 with kernel release 3.11. The host machine has 16GB RAM, 160GB Hard Disk, x86_64 architecture, and 8 processors running Ubuntu 14.04 with kernel release 3.13.

Figure 3 shows an example of the input format a developer needs to provide. By default, Bear tests all system calls with all strategies, with 1.5 times of the estimated time to complete as timeout in case any test hangs in the middle of a run. In the example given in Figure 3, `g++` is tested by generating optimized code on a Solaris machine with warnings. After

Input Format

```
$ g++ -Wall -O -mv8 myprog.C -o myprog
Syscall: All          Timeout: 60
Strategy: All         Threshold: 30%
File_syscall: No      File_thresh: 0%
Net_syscall: No       Net_thresh: 0%
...
Sig_syscall: No       Sig_thresh: 0%
```

Output Format

Target: **g++**

System call	Odds Ratio	Count
read	3.53021	10
write	1.22807	16
...	...	...

Strategy	Odds Ratio
failMem	0.35442

Fig. 3: Format of Bear input and output.

the analysis, Bear outputs system calls and strategies ranked by impact. The number of times each system call is invoked (*Count*) is also provided.

As discussed in Section IV, for each collection of program runs we performed a correlation and a regression analysis. The next subsections describe each one of these analyses and how they were evaluated. In our evaluation, we considered the significance level $\alpha = 0.05$ (see section IV). The evaluation is organized around the following questions:

1) Which perturbation strategies are the most and least impactful on abnormal program execution? Does it vary based on threshold?
2) Which system calls are the most and least sensitive to perturbation? Does it vary based on perturbation threshold?
3) How does the strategy impact and system call sensitivity vary based on the program type (processor, disk, network, and encoding)?
4) Will the result differ when applications are executed with different workloads?

A. Correlation Analysis

As illustrated in Figure 2, the first analysis phase in Bear is a correlation analysis with a chi-squared test to discover whether the program outcome is independent of applying perturbation strategies on system calls. Table II and III illustrate the chi-square test results between abnormal execution outcome and system calls/strategies. df represents the degrees of freedom and, in our example, equals the number of strategies/system calls minus one. We considered all applications in categories processor and encoding as CPU-bound, and all applications in categories network or disk as I/O-bound.

As discussed in section IV, the evidence to reject the null hypothesis is a chi-square statistic value higher than that in the chi-square distribution table [18]. For system calls ($df = 20$), the chi-squared distribution table shows a value of 39.997, which is lower than every X-squared value in Table II. For strategies ($df = 9$), the chi-squared distribution table shows a value of 18.548, which is also lower than every X-squared value in Table III. This leads us to reject the null hypothesis of independence and conclude that there is an association between a program execution outcome and a perturbation system call, and likewise on a perturbation strategy.

	X-squared value	df	p-value
All programs	262.39	20	<0.0001
CPU-bound	48.732	20	0.0003355
IO-bound	486.21	20	<0.0001

TABLE II: Chi-square result for *system call* and program execution outcome.

	X-squared value	df	p-value
All programs	66.428	9	<0.0001
CPU-bound	42.667	9	<0.0001
IO-bound	112.02	9	0.0017

TABLE III: Chi-square result for *strategy* and program execution outcome.

To understand the strength of the association between these variables, we did follow-up tests using the Cramer's V method. The tests produced association levels from 15% to 35%, which are acceptable and even desirable based on Cramer's V criteria [19] (see section IV). Given these results we could proceed to the next phase in our evaluation: logistic regression, which allows us to better understand the influence of system call and strategy on the program outcome.

B. Regression Analysis

This section describes the logistic regression model generated by Bear.

1) Strategy Impact on Program Outcome: Bear uses two types of program outcomes: normal and abnormal. We initially classified the possible program outcomes into four levels: normal, crash, abort, and segfaults. During the analysis of 113 programs, we observed that the number of aborts and segfaults was 1-2 orders of magnitude smaller than the number of normal executions and crashes, which would decrease the accuracy of the statistical model. Coupled with controversial opinions on how to classify normal and crashes [20], this prompted us to consider crashes, aborts, and segfault as a single category of program outcome: abnormal. In other words, result **normal** referred to a correct execution or a graceful exit of a program and result **abnormal** referred to all the other results.

Figure 4 shows the impact of perturbation strategies as predictors on program outcome for all programs and for the thresholds of 10%, 50% and 90%. With $p<0.05$, every strategy except `failNoti` and `chUid` are statistically significant in predicting an abnormal program outcome. Odds ratios on the x-axis show the odds of a strategy causing an abnormal program outcome when compared to a reference strategy. The reference strategy doesn't make a difference on the analysis result. We selected *nullMem* as the reference strategy and plotted how much more or less likely (odds ratio) the other strategies are in causing an abnormal outcome compared to *nullMem*, the most significant strategy in predicting an abnormal program outcome.

For example, for the perturbation threshold of 90%, strategy *failAcc* has an odds ratio of 0.5 ($p<0.001$) when compared

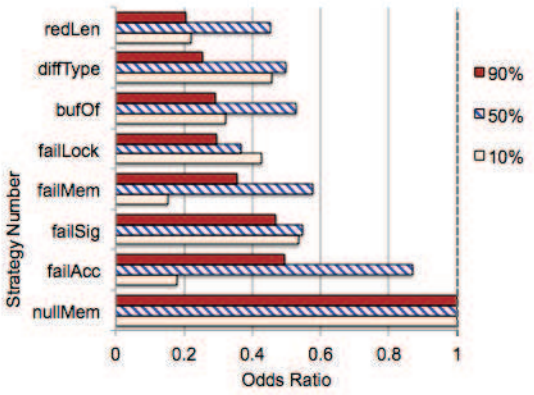

Fig. 4: The impact of perturbation strategies in predicting abnormal program outcome, tested on **all** programs with thresholds of 10%, 50% and 90%. Reference strategy is *nullMem*. Odds ratio shows how much more or less likely a strategy is to cause an abnormal program outcome compared to the reference strategy. Absence of `chUid` and `failNoti` means that they are not significant in predicting an abnormal execution.

with *nullMem*, which means that it is 50% less likely to cause an abnormal program outcome than *nullMem* for the same threshold. Notice that the odds ratios vary depending on the perturbation threshold. For example, *diffType* is more likely to cause an abnormal program outcome for 10% and 50% perturbation threshold than for 90%.

On threshold 50% and 90%, the impact of strategies follows a similar trend, with `nullMem` `failAcc` and `failSig` as the three most impactful strategies and `failLock` `diffType` and `redLen` as the three least impactful strategies. The impact of strategies on threshold 10% differed by some degree. When the threshold increases from 50% to 90%, the overall impact of the strategies decreases compared to reference strategy `nullMem`, which shows that the impact of `nullMem` increases faster than that of the rest of strategies. Also, the impact of `failSig` and `failLock` increases faster than that of buffer and memory-related strategies such as `failAcc`, `failMem`, `bufOf`, `diffType` and `redLen`.

The absence of strategies `chUid` and `failNoti` in Figure 4 does not necessarily mean that they have no impact on an abnormal program execution. They are just not statistically significant in predicting an abnormal result under the defined level $\alpha = 0.05$. In other words, their impact is relatively little compared to the other strategies.

We evaluated the tested programs execution based on whether the application was CPU or I/O-bound. We considered all applications in categories processor and encoding as CPU-bound, and all applications in categories network or disk as I/O-bound.

Figures 5 and 6 show the impact of the strategies on program outcome when we analyze I/O-bound and CPU-bound programs separately. Figure 5 shows that only five strategies related to buffer and memory demonstrate statistical significance in predicting an abnormal program outcome for I/O-bound pro-

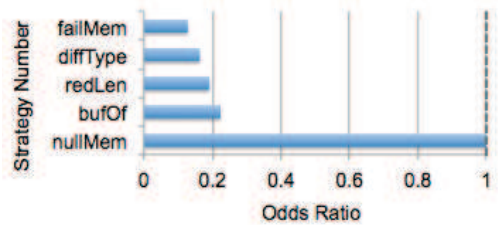

Fig. 5: Impact of perturbation strategies in predicting abnormal program outcome, tested on **I/O-bound** programs. Each strategy is compared with reference strategy *nullMem*.

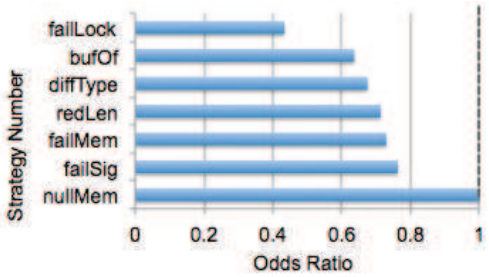

Fig. 6: The impact of perturbation strategies in predicting abnormal program execution, tested on **CPU-bound** programs. Each strategy is compared with reference strategy *nullMem*.

grams. `nullMem` still has the strongest impact while `bufOf`, `redLen`, and `diffType` show similar impacts at around 20% of `nullMem`. For CPU-bound programs, the result shows some differences as displayed in Figure 6. Besides the five statistically significant strategies in I/O-bound programs, signal-related strategies such as `failLock` and `failSig` are also significant in CPU-bound programs. `nullMem` still has the strongest impact, and the rest of the strategies are at about 50% to 70% of its impact. `failAcc` does not appear in Figure 5 and 6 because of the *complete separation* statistical phenomenon that occurred after we separated the data from CPU and IO-bound programs. *Complete separation* occurs when a linear combination of the predictors yields a perfect prediction of the response, which skews the model and reduces its overall predictive strength. In our experiment, any system call/strategy presenting a number of normal outcomes 20 times more than the number of abnormal outcomes is considered affected by complete separation.

The lesson learned is that `nullMem` is the highest impactful misbehavior for predicting an abnormal execution in both I/O-bound (disk and network) applications and CPU-bound (processor and encoding) applications. Buffer overflow, value outside domain, unchecked return length, and memory leak should be carefully considered during software development. CPU-bound software is especially sensitive to signal-related errors. In case of limited resource (time, performance hit limit, man hours) for testing/verification, naked notify and privilege degradation can be given the lowest priority as they bring the least impact on program execution.

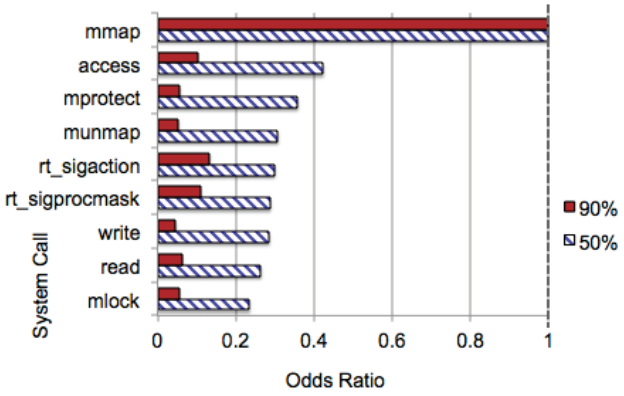

Fig. 7: Top nine most impactful system calls on predicting abnormal program outcomes for **all** programs, compared with reference system call `mmap`.

2) System Call Impact on Program Execution: At threshold 10%, every system call makes equal little impact in predicting an abnormal program outcome. Thus in this section, we only consider system call sensitivity with the thresholds of 50% and 90%. At these threshold levels, every system call is statistically significant in predicting an abnormal program outcome. Figure 7 shows the nine most impactful system calls compared to the reference system call `mmap` on abnormal program outcome. The likelihood of an abnormal program outcome is again demonstrated by using odds ratios on the x-axis. `mmap` is chosen as the reference system call since it is the most statistically significant in predicting an abnormal program outcome. This finding corroborates our previous finding that `failMem` and `nullMem` are the two most impactful strategies in causing abnormal program execution.

At threshold 50%, system call `access` is approximately 60% less sensitive than `mmap` and is the second-most sensitive for abnormal execution. A failure in `access` prevents a program from accessing a library and can abruptly cause its termination. The other system calls demonstrate a relatively equal significance. Network-related system calls such as `sendto` and `recvfrom` are not in the top nine most significant list, which can be attributed to their robustness against perturbation or the presence of robust end-to-end checks and retry mechanisms over network layer. At threshold 90%, every system call presents only 10% to 15% of `mmap`'s sensitivity. This happens because the sensitivity of `mmap` increases faster than the other system calls when the threshold increases.

Figures 8 and Figure 9 evaluate the I/O-bound and CPU-bound subsets respectively. In I/O-bound programs, only five system calls are statistically significant in predicting an abnormal outcome. `mmap` was removed from the analysis model for I/O-bound programs due to the *complete separation* problem. Any perturbation on `mmap` for all thresholds almost always caused an abnormal program outcome, and adding a variable where the outcome is already known skews the model and reduces its overall predictive strength.

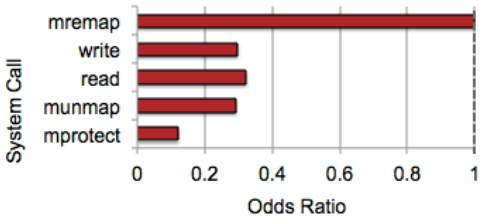

Fig. 8: Statistically significant system calls on predicting abnormal program outcomes for **IO-bound** programs, compared with reference system call `mremap`.

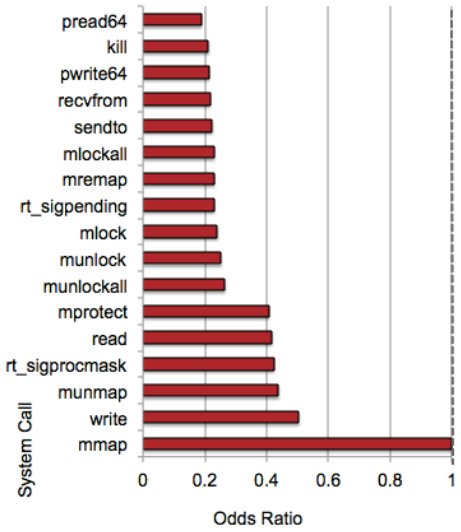

Fig. 9: Statistically significant system calls on predicting abnormal program outcomes for **CPU-bound** programs, compared with reference system call `mmap`.

Thus after eliminating `mmap` from the model, `mremap` becomes the most sensitive system call and works as the reference system call in Figure 8. Again, no network-specialized or signal-related system calls show up in the figure, corroborating the result in Figure 7 that no network-specialized system calls are highly significant and the result in Figure 5 that no signal-related strategies are highly significant.

Figure 9 listed all the system calls that are statistically significant for an abnormal CPU-bound program outcome compared with reference system call `mmap`. Generic system calls (such as `write` and `read`), memory-related system calls (such as `munmap` and `mprotect`) and signal-related system calls (such as `rt_sigprocmask`) have more than 40% of the sensitivity of `mmap`. Network-related system calls, such as `sendto recvfrom`, show lower sensitivity at around 20% of `mmap`. System calls such as `readv/write` and `preadv/pwritev` with a vector buffer do not appear in Figure 9, while `read/write` and `pread64/pwrite64` with a general buffer do, indicating that system calls with a vector buffer are less impactful in predicting an abnormal program execution. Notice that even though some system calls present equal sensitivity, such as `sendto` and `mlockall`,

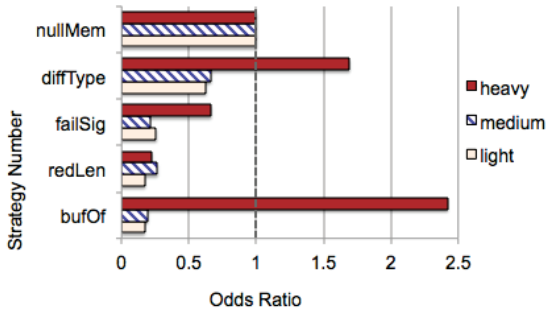

Fig. 10: Impactful strategies on predicting abnormal program outcomes on three workloads, compared with reference strategy `nullMem`.

the frequency at which they are called is totally different. Developer and testers should balance their time, not only based on how severe the impact on the system call might be, but also on how frequently the system call is invoked.

3) Workload Impact on Program Outcome: SPEC CPU2006 and Phoronix Test Suite provide applications that can be configured to run under different workloads. Twenty-six applications are selected because they can be installed successfully on our experiment virtual machine. The workloads contain three levels: light, medium, and heavy, which correspond to test, train, and ref levels for SPEC CPU2006, and first, middle-most, and last level in Phoronix Test Suite.

Figure 10 shows the statistically significant strategies in predicting abnormal program outcome on three workloads compared with reference strategy `nullMem`. From Figure 10, we can see that the results for light and medium workloads follow the same trend, with `nullMem` and `diffType` highly impactful and `failSig`, `redLen` and `bufOf` less impactful. However, when workload increases from medium to heavy, `bufOf` rapidly grows the impact to 10 times of the original, and `diffType` and `failSig` double their impacts. Figure 11 demonstrates the sensitive system calls on predicting abnormal outcome for the three different workloads. Every system call in the figure has 10% to 20% sensitivity of `mmap`'s sensitivity and presents equal impact on three workloads, indicating that the workload does not influence the system call sensitivity on predicting abnormal program outcome.

This tells the developers that regardless of the system call type, buffer overflows and value outside domain bugs should be taken seriously when heavy workloads are used.

C. Summary and Recommendations for Developers

Our results showed that for CPU-bound applications, the four most sensitive system calls are `mmap`, `write`, `munmap` and `rt_sigprocmask`. While for I/O-bound applications, the four most sensitive system calls are `mremap`, `write`, `read` and `munmap`. For both I/O-bound and CPU-bound applications, null dereferencing and buffer overflow strategies

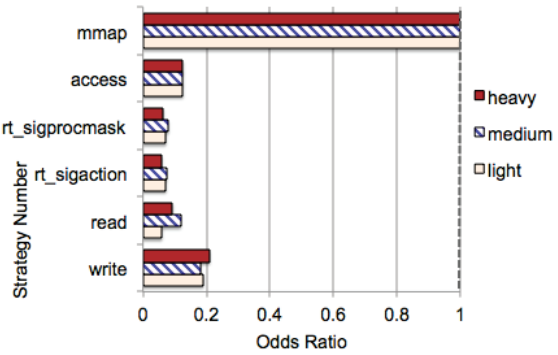

Fig. 11: Statistically sensitive system calls on predicting abnormal program outcomes on different workloads, compared with reference system call `mmap`.

are the two most severe in predicting an abnormal program outcome. For CPU-specific applications, signal-related errors are of the equal importance. The two least impactful strategies are naked notify in method and privilege degradation, and the rest of the strategies are of similar impact.

Surprisingly, network-specialized system calls (`read` and `write` excluded) didn't show high impact on abnormal execution, taking into account their wide use in programs and APIs, and this should be attributed to the effectiveness of a perfect end-to-end checking and retry mechanism over network layer.

When workload is heavy, the impact of buffer overflow on abnormal execution doubled five times than medium workload and far exceeded the impact of null dereferencing. The same thing happens on wrong-type buffer strategy, which doubles the impact on heavy workload. On light and medium workload, strategy impact and system call sensitivity remains invariant for all applications.

When resources are limited (time, man hours, performance), developers should focus on buffer type and return length checking, considering how commonly used `read` and `write` system calls are, especially when application workload is heavy. One strategy developers should take is to use system calls that are specialized and that use vector arrays in a struct parameter, rather than a standard array in the parameter list. Moreover, given a choice of more than one system call for a particular functionality developers should choose those with a larger parameter list. Null dereferencing is a severe problem as well and almost the hardest to debug when a segmentation fault occurs. Therefore, failure-oblivious computing can be a promising way for saving developers and testers from memory bugs.

VI. Related Work

This paper intersects the areas of fuzz testing, fault-tolerance, failure-oblivious computing, and unpredictability and diversity in computer systems. In this section we discuss relevant work in these areas.

Fuzz Testing: Fuzz testing is an effective way to discover coding errors and security loopholes in software, operating systems, and networks by testing applications against invalid, unexpected, or random data inputs. Miller et al. [21] first proposed fuzz testing as an inexpensive mechanism to generate additional software tests. The authors later extended the work [22] to identify missed return code checks from crucial calls, such as memory allocation. Many additional fuzz testing approaches have been proposed [23]–[25]. Trinity [26], for example, randomizes system call parameters to test the validation of file descriptors, and found real bugs [27], including bugs in the Linux kernel [28]–[30]. White-box fuzzy testers [31]–[34] were also proposed to increase the coverage of test inputs by leveraging symbolic execution and dynamic test generation. For instance, KLEE [33] uses symbolic execution and a model of system call behaviors provided by a user to generate high-coverage test cases. BALLISTA [20] tests the data type robustness of the POSIX system call interface in a scalable way, by defining 20 data types for testing 233 system calls of the POSIX standard. Bear can also be thought as a fuzz tester at the OS system call API and contributes statistical techniques to understand how sensitive an application is to a particular type of OS misbehavior.

Fault Injection: Fault injection is an important method for generating test cases in fuzz testing. Through fault injection, researchers are able to study fault propagation [35] and develop flexible and robust software and systems [36]–[38]. Kanawati and Abraham provide a methodology and guidelines for the design of flexible software, based on their experience with the fault injection tool FERRARI [36]. Fault injection has been applied to a number of abstractions. DOCTOR [37] for example, supports memory faults, CPU faults, and communication faults. FINE [35] traces execution flow and key variables through the UNIX kernel via hardware-induced software errors and kernel software faults injection. A recent survey on assessing dependability with software fault injection [39] provides a comprehensive overview of the state of the art fault injection approaches to fit the goals of researchers and practitioners. LFI tool [40] injects errors in library-calls, in order to identify error handling faults that arise from misunderstanding of library APIs, and from poor portability across different OSes. Other possible forms of fault injection are code mutations and data interface corruptions [41].

Bear also works by injecting faults (perturbations) in the execution of software at the system call level. Even though this fault injection can be leveraged in fuzzy testers, Bear's goal is to statistically understand program sensitivity to different types of faults on different types of system calls.

Failure-Oblivious Computing: Failure-oblivious computing allows a system or program to continue execution in spite of errors. Rinard *et al.* [42] published a classic paper in the area that introduced a C compiler to insert checks to dynamically detect invalid memory accesses. On errors, instead of terminating the program or throwing an exception, the program would discard invalid writes and manufacture values

to return for invalid reads, enabling the program to continue its execution. ASSURE [43] introduces rescue points in software (discovered via fuzzing) to recover it from unknown faults, while maintaining both system integrity and availability. Bear's goals are complementary to failure-oblivious approaches. By determining the most impactful types of perturbations and the most sensitive system calls, it can help failure-oblivious approaches to better target checks and rescue points, greatly improving the effectiveness and the high performance overhead of such approaches.

Unpredictability and Diversity: A major new trend in secure systems is toward mutual distrust between applications and operating systems [6], [7], facilitated by technologies such as Intel SGX [4]. SGX offers features such as protecting application memory from the OS, as well as a hardware-attested, cryptographic hash of the launched code. OSes are commonly compromised [44], and are often considered untrustworthy by users—especially in cloud computing environments, where software components may be provided by a third party. Iago attacks [3], for example, compromise a process execution by manipulating system call return values, illustrating the importance of deeper analysis of how resilient an application is to unexpected OS behavior.

Several projects mitigate buffer overflows and other memory errors by randomizing system call mappings, global library entry points, stack placement, stack direction, and heap placement, often in conjunction with running multiple versions in parallel to detect divergence [45], [46]. The Synthetix project [47] specializes code dynamically using automatic compiler analysis and programmer annotations, primarily to improve performance; specialization has been proposed as a mechanism to block attacks [48]. Program slicing has also been used to bound the cost and complexity of automatic diversification [49]. Containing unpredictability has an effect of increasing server availability. Rinard et al. [50] demonstrated how failure-oblivious computing can enable servers to execute through memory errors without memory corruption. By converting unanticipated and dangerous execution paths into invalid inputs in the error-handling logic, the server continued on processing subsequent requests, and largely extended the range of requests processed. Compiler diversification and instruction randomization techniques [51] are complementary and orthogonal to Bear, as software generated with such techniques will be more resilient to random and adversarial perturbations. Further, Bear can provide insights about which functionalities are the best candidates for perturbation as a way to create diversity in the system. For example, proposals like those of Sun *et al.* [52], [53] can select approaches for OS unpredictability that minimize the impact on benign applications, while still generating diverse runs of the same system.

VII. Conclusions and Future Work

This paper introduced Bear, a framework for statistical analysis of program sensitivity to OS unpredictability that aids developers in writing programs that are resilient to OS misbehavior. Bear's evaluation produced important insights for developers.

As expected, perturbations like null dereferencing and buffer overflow are the two most significant strategies in predicting abnormal program outcomes, while the two least impactful strategies are naked notify in method and privilege degradation. For CPU-specific applications, signal-related perturbations are also highly impactful. For system call sensitivity, network-specialized system calls (`read` and `write` excluded) did not show high impact on abnormal execution, taking into account their wide use in programs and APIs, and this should be attributed to the effectiveness of a perfect end-to-end checking and retry mechanism over network layer. Generic system calls are more sensitive than specialized system calls—for example, socket-related system calls `read/write` are twice the sensitivity of `sendto/recvfrom`. System calls having a general buffer are more sensitive than those having a vector buffer parameter (e.g. `read/write` is in the sensitive list but `readv/writev` is not). System calls having fewer number of parameters specified are more sensitive than those having a larger number of parameters list (e.g. `access` is in the sensitive list but `faccessat` is not).

Developers should be careful on handling buffer type and return length of OS system calls, considering how commonly used `read` and `write` system calls are, especially when application workload is heavy. Developers and testers should balance their time not only based on how severe the impact of a system call or a strategy, but also on how frequently it is invoked. If resources are limited, failure-oblivious computing can be a promising way for saving developers and testers from memory bugs. Given an option of more than one system calls for the same functionality, developers should select system calls that are specialized, taking a buffer of vector inside a parameter struct and having the larger parameter list.

With slight changes, Bear's framework can be applied for all OSes because most of the system calls in the perturbation set are portable/generic. The future work will include finer granularity on the abnormal execution results.

VIII. Acknowledgment

We thank the ISSRE anonymous reviewers for their insightful comments. This research is supported by NSF grant CNS-1149730, CNS-1228839 and DGE-1303211.

References

[1] Y. Zhuang, E. Gessiou, S. Portzer, F. Fund, M. Muhammad, I. Beschastnikh, and J. Cappos, "Netcheck: Network diagnoses from blackbox traces," in *USENIX Symposium on Networked Systems Design and Implementation*, 2014.

[2] J. Rasley, E. Gessiou, T. Ohmann, Y. Brun, S. Krishnamurthi, and J. Cappos, "Detecting latent cross-platform api violations," in *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2015.

[3] S. Checkoway and H. Shacham, "Iago attacks: Why the system call api is a bad untrusted rpc interface," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013, pp. 253–264.

[4] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using innovative instructions to create trustworthy software solutions," in *Workshop of Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[5] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: secure applications on an untrusted operating system," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013, pp. 265–278.

[6] J. Criswell, N. Dautenhahn, and V. Adve, "Virtual ghost: Protecting applications from hostile operating systems," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014, pp. 81–96.

[7] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 267–283.

[8] "System calls in perturbation set." [Online]. Available: https://docs.google.com/document/d/1E7futmb_XisCHJkh2VTJtRxswR_uSqmezYVWSl9Tgyg/edit?usp=sharing

[9] V. Vipindeep and P. Jalote, "List of common bugs and programming practices to avoid them," *Electronic, March*, 2005.

[10] "The r project for statistical computing https://www.r-project.org/."

[11] F. Gravetter and L. Wallnau, *Statistics for the Behavioral Sciences*, 8th ed. Wadsworth/Thomson Learning, 2009.

[12] A. Agresti, *Categorical Data Analysis*. John Wiley & Sons, 2002.

[13] "Linear regression models http://people.duke.edu/~rnau/regex3.htm."

[14] "Coreutils - gnu core utilities http://www.gnu.org/software/coreutils/coreutils.html."

[15] "Spec cpu 2006 https://www.spec.org/cpu2006/."

[16] "The phoronix test suite http://www.phoronix-test-suite.com/."

[17] J. H. McDonald, *Handbook of Biological Statistics*, 3rd ed. Sparky House Publishing, 2014.

[18] "Chi square distribution table http://sites.stat.psu.edu/~mga/401/tables/Chi-square-table.pdf."

[19] "Cramer's v measures of association http://groups.chass.utoronto.ca/pol242/Labs/LM-3A/LM-3A_content.htm."

[20] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz, "Comparing operating systems using robustness benchmarks," in *Reliable Distributed Systems, 1997. Proceedings., The Sixteenth Symposium on*. IEEE, 1997, pp. 72–79.

[21] B. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl, "Fuzz revisited: A re-examination of the reliability of UNIX utilities and services," Tech. Rep., 1995.

[22] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the Association for Computing Machinery*, vol. 33, no. 12, pp. 32–44, 1990.

[23] C. Miller and Z. N. J. Peterson, "Analysis of Mutation and Generation-Based Fuzzing," Independent Security Evaluators, Tech. Rep., Mar. 2007.

[24] U. Kargén and N. Shahmehri, "Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing," *Target*, vol. 101, no. 1101, p. 1011, 2015.

[25] Z. Zhang, Q.-Y. Wen, and W. Tang, "An efficient mutation-based fuzz testing approach for detecting flaws of network protocol," in *Computer Science & Service System (CSSS), 2012 International Conference on*. IEEE, 2012, pp. 814–817.

[26] "Lca: The trinity fuzz tester." [Online]. Available: https://lwn.net/Articles/536173/

[27] "Bugs found by trinity." [Online]. Available: http://codemonkey.org.uk/projects/trinity/bugs-fixed.php

[28] B. Garn and D. E. Simos, "Eris: A tool for combinatorial testing of the linux system call interface," in *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on*. IEEE, 2014, pp. 58–67.

[29] A. Kurmus, "Kernel self-protection through quantified attack surface reduction."

[30] V. M. Weaver and D. Jones, "perf fuzzer: Targeted fuzzing of the perf event open () system call."

[31] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing," in *NDSS*, vol. 8, 2008, pp. 151–166.

[32] D. Molnar, X. C. Li, and D. Wagner, "Dynamic test generation to find integer bugs in x86 binary linux programs." in *USENIX Security Symposium*, vol. 9, 2009.

[33] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224.

[34] N. Tillmann and J. De Halleux, "Pex–white box test generation for. net," in *Tests and Proofs*. Springer, 2008, pp. 134–153.

[35] W.-L. Kao, R. K. Iyer, and D. Tang, "Fine: A fault injection and monitoring environment for tracing the unix system behavior under faults," *Software Engineering, IEEE Transactions on*, vol. 19, no. 11, pp. 1105–1118, 1993.

[36] G. Kanawati, N. Kanawati, J. Abraham *et al.*, "Ferrari: A flexible software-based fault and error injection system," *Computers, IEEE Transactions on*, vol. 44, no. 2, pp. 248–260, 1995.

[37] S. Han, K. G. Shin, H. Rosenberg *et al.*, "Doctor: An integrated software fault injection environment for distributed real-time systems," in *Computer Performance and Dependability Symposium, 1995. Proceedings., International*. IEEE, 1995, pp. 204–213.

[38] J. Carreira, H. Madeira, J. G. Silva *et al.*, "Xception: Software fault injection and monitoring in processor functional units," *Dependable Computing and Fault Tolerant Systems*, vol. 10, pp. 245–266, 1998.

[39] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing dependability with software fault injection: A survey," *ACM Comput. Surv.*, vol. 48, no. 3, pp. 44:1–44:55, Feb. 2016.

[40] P. D. Marinescu and G. Candea, "Efficient testing of recovery code using fault injection," *ACM Trans. Comput. Syst.*, vol. 29, no. 4, pp. 11:1–11:38, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2063509.2063511

[41] A. Lanzaro, R. Natella, S. Winter, D. Cotroneo, and N. Suri, "An empirical study of injected versus actual interface errors," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. ACM, pp. 397–408.

[42] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, Jr., "Enhancing server availability and security through failure-oblivious computing," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, pp. 21–21.

[43] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis, "Assure: Automatic software self-healing using rescue points," *SIGPLAN Not.*, vol. 44, no. 3, pp. 37–48, Mar. 2009.

[44] A. Srivastava and J. Giffin, "Efficient monitoring of untrusted kernel-mode execution," *NDSS*, 2011.

[45] M. Chew and D. Song, "Mitigating buffer overflows by operating system randomization," University of California, Berkeley, Tech. Rep., 2002.

[46] E. D. Berger and B. G. Zorn, "DieHard: Probabilistic Memory Safety for Unsafe Languages," *PLDI*, pp. 158–168, June 2006.

[47] D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, A. Goel, P. Wagle, C. Consel, G. Muller, and R. Marlet, "Specialization tools and techniques for systematic optimization of system software," *ACM Trans. Comput. Syst.*, vol. 19, no. 2, pp. 217–251, May 2001.

[48] C. Pu, A. P. Black, C. Cowan, J. Walpole, and C. Consel, "A specialization toolkit to increase the diversity of operating systems," in *Proceedings of the ICMAS Workshop on Immunity-Based Systems*, 1996.

[49] J. P. Sterbenz and P. Kulkarni, "Diverse infrastructure and architecture for datacenter and cloud resilience," in *Computer Communications and Networks (ICCCN), 2013 22nd International Conference on*. IEEE, 2013, pp. 1–7.

[50] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and J. William S. Beebee, "Enhancing Server Availability and Security through Failure-Oblivious Computing," *OSDI*, 2004.

[51] D. A. Holland, A. T. Lim, and M. I. Seltzer, "An architecture a day keeps the hacker away," *SIGARCH Comput. Archit. News*, vol. 33, no. 1, pp. 34–41, Mar. 2005.

[52] R. Sun, D. E. Porter, D. Oliveira, and M. Bishop, "The case for less preditable operating system behavior," in *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[53] R. Sun, M. Bishop, N. C. Ebner, D. Oliveira, and D. E. Porter, "The Case for Unpredictability and Deception as OS Features," *USENIX ;login*, 2015.