2017 IEEE Symposium on Security and Privacy

NEZHA: Efficient Domain-Independent Differential Testing

Theofilos Petsios*, Adrian Tang*, Salvatore Stolfo, Angelos D. Keromytis and Suman Jana
Department of Computer Science
Columbia University, New York, USA
{theofilos, atang, sal, angelos, suman}@cs.columbia.edu

***Abstract*—Differential testing uses similar programs as cross-referencing oracles to find semantic bugs that do not exhibit explicit erroneous behaviors like crashes or assertion failures. Unfortunately, existing differential testing tools are domain-specific and inefficient, requiring large numbers of test inputs to find a single bug. In this paper, we address these issues by designing and implementing NEZHA, an efficient input-format-agnostic differential testing framework. The key insight behind NEZHA's design is that current tools generate inputs by simply borrowing techniques designed for finding crash or memory corruption bugs in individual programs (e.g., maximizing code coverage). By contrast, NEZHA exploits the behavioral asymmetries between multiple test programs to focus on inputs that are more likely to trigger semantic bugs. We introduce the notion of δ-diversity, which summarizes the observed asymmetries between the behaviors of multiple test applications. Based on δ-diversity, we design two efficient domain-independent input generation mechanisms for differential testing, one gray-box and one black-box. We demonstrate that both of these input generation schemes are significantly more efficient than existing tools at finding semantic bugs in real-world, complex software.**

NEZHA's average rate of finding differences is 52 times and 27 times higher than that of Frankencerts and Mucerts, two popular domain-specific differential testing tools that check SSL/TLS certificate validation implementations, respectively. Moreover, performing differential testing with NEZHA results in 6 times more semantic bugs per tested input, compared to adapting state-of-the-art general-purpose fuzzers like American Fuzzy Lop (AFL) to differential testing by running them on individual test programs for input generation.

NEZHA discovered 778 unique, previously unknown discrepancies across a wide variety of applications (ELF and XZ parsers, PDF viewers and SSL/TLS libraries), many of which constitute previously unknown critical security vulnerabilities. In particular, we found two critical evasion attacks against ClamAV, allowing arbitrary malicious ELF/XZ files to evade detection. The discrepancies NEZHA found in the X.509 certificate validation implementations of the tested SSL/TLS libraries range from mishandling certain types of `KeyUsage` extensions, to incorrect acceptance of specially crafted expired certificates, enabling man-in-the-middle attacks. All of our reported vulnerabilities have been confirmed and fixed within a week from the date of reporting.

I. INTRODUCTION

Security-sensitive software must comply with different high-level specifications to guarantee its security properties. Any semantic bug that causes deviations from these specifications might render the software insecure. For example, a malware detector must parse input files of different formats like ELF (the default executable format in Linux/Unix-based systems), PDF, or XZ (a popular archive format), according to their respective specifications, in order to accurately detect malicious content hidden in such files [41]. Similarly, SSL/TLS implementations must validate X.509 certificates according to the appropriate protocol specifications for setting up a secure connection in the presence of network attackers [24], [33].

However, most semantic bugs in security-sensitive software do not display any explicitly erroneous behavior like a crash or assertion failure, and thus are very hard to detect without specifications. Unfortunately, specifications, even for highly critical software like SSL/TLS implementations or popular file formats like ELF, are usually documented informally in multiple sources such as RFCs and developer manuals [10]–[18], [20], [62], [63]. Converting these informal descriptions to formal invariants is tedious and error-prone.

Differential testing is a promising approach towards overcoming this issue. It finds semantic bugs by using different programs of the same functionality as cross-referencing oracles, comparing their outputs across many inputs: any discrepancy in the programs' behaviors on the same input is marked as a potential bug. Differential testing has been used successfully to find semantic bugs in diverse domains like SSL/TLS implementations [24], [32], C compilers [65], and JVM implementations [31]. However, all existing differential testing tools suffer from two major limitations as described below.

First, they rely on domain-specific knowledge of the input format to generate new test inputs and, therefore, are brittle and difficult to adapt to new domains. For instance, Frankencerts [24] and Mucerts [32] incorporate partial grammars for X.509 certificates and use domain-specific mutations for input generation. Similarly, existing differential testing tools for C compilers, Java virtual machines, and malware detectors, all include grammars for the respective input format and use domain-specific mutations [31], [41], [65].

Second, existing differential testing tools are inefficient at finding semantic bugs, requiring large numbers of inputs to be tested for finding each semantic bug. For example, in our experiments, Frankencerts required testing a total of 10 million inputs to find 10 distinct discrepancies, starting from a corpus of $100,000$ certificates. Mucerts, starting from the same $100,000$ certificates, reported 19 unique discrepancies, using $2,660$ optimized certificates it generated from the corpus, but required six days to do so.

In this paper, we address both the aforementioned prob-

*Joint primary student authors.

DOI 10.1109/SP.2017.27

lems by designing and implementing NEZHA[1], a differential testing tool that uses a new domain-independent approach for detecting semantic bugs. NEZHA does not require any detailed knowledge of the input format, but still significantly outperforms existing domain-specific approaches at finding new semantic bugs.

Our key observation is that existing differential testing tools ignore asymmetries observed across the behaviors of all tested programs, and instead generate test inputs simply based on the behaviors of individual programs in isolation. For instance, Mucerts try to maximize code coverage solely on a single program (e.g., OpenSSL) to generate inputs. However, this approach cannot efficiently find high numbers of unique semantic bugs since all information on the differences each input might introduce *across* the tested programs is ignored. As a result, despite using domain-specific guided input generation, existing differential testing tools are inefficient. In this paper, we address this issue by introducing the notion of δ-diversity —a method for summarizing the behavioral asymmetries of the tested programs. Under δ-diversity guidance, these asymmetries can be expressed in different ways, examining each individual program's behavior in either a black-box (based on program log/warning/error messages, program outputs, *etc.*) or gray-box (e.g., program paths taken during execution) manner.

The main difference between our approach and prior differential testing tools is that we generalize the tracking of guidance information across all tested programs, examining their behaviors relative to each other, not in isolation, for guided input generation. For example, if two test programs execute paths p_1 and p_2, respectively, for the same input, a "δ-diversity-aware" representation of the execution will consist of the tuple $\langle p_1, p_2 \rangle$. Our guidance mechanism for input generation is designed to maximize δ-diversity, i.e., the number of such tuples. We demonstrate in Section V that our scheme is significantly more efficient at finding semantic bugs than using standalone program testing techniques. We compare NEZHA with Frankencerts, Mucerts, as well as with two state-of-the-art fuzzers, namely AFL [66] and libFuzzer [4]. In our testing of certificate validation using major SSL/TLS libraries, NEZHA finds 52 times, 27 times, and 6 times more unique semantic bugs than Frankencerts, Mucerts, and AFL respectively.

NEZHA is input-format-agnostic and uses a set of initial seed inputs to bootstrap the input generation process. Note that the seed files themselves do not need to trigger any semantic bugs. We empirically demonstrate that NEZHA can efficiently detect subtle semantic differences in large, complex, real-world software. In particular, we use NEZHA for testing: (i) ELF and XZ file parsing in two popular command-line applications and the ClamAV malware detector, (ii) X.509 certificate validation across six major SSL/TLS libraries and (iii) PDF parsing/rendering in three popular PDF viewers. NEZHA discovered 778 distinct discrepancies across all tested families of applications, many of which constitute previously unknown security vulnerabilities. For example, we found two evasion attacks against ClamAV, one for each of the ELF and XZ parsers. Moreover, NEZHA was able to pinpoint 14 unique differences even among forks of the same code base like the OpenSSL, LibreSSL, and BoringSSL libraries.

In summary, we make the following contributions:

- We introduce the concept of δ-diversity, a novel scheme that tracks relative behavioral asymmetries between multiple test programs to efficiently guide the input generation process of differential testing.
- We build and open-source NEZHA, an efficient, domain-independent differential testing tool that significantly outperforms both existing domain-specific tools as well as domain-independent fuzzers adapted for differential testing.
- We demonstrate that NEZHA is able to find multiple previously unknown semantic discrepancies and security vulnerabilities in complex real-world software like SSL/TLS libraries, PDF viewers, and the ClamAV malware detector.

The rest of the paper is organized as follows. We provide a high-level overview of our techniques with a motivating example in Section II. Section III details our methodology. We describe the design and implementation of NEZHA in Section IV and present the evaluation results of our system in Section V. We highlight selected case studies of the bugs NEZHA found in Section VI. Finally, we discuss related work in Section VII, future work in Section VIII, and conclude in Section X.

II. OVERVIEW

A. Problem Description

Semantic bugs are particularly dangerous for security-sensitive programs that are designed to classify inputs as either valid or invalid according to certain high-level specifications (e.g., malware detectors parsing different file formats or SSL/TLS libraries verifying X.509 certificates). If an input fails to conform to these specifications, such programs typically communicate the failure to the user by displaying an error code/message. For the rest of the paper, we focus on using differential testing to discover program discrepancies in this setting, i.e., where at least one test program validates and accepts an input and another program with similar functionality rejects the same input as invalid. Attackers can exploit this class of discrepancies to mount evasion attacks on malware detectors. They can also compromise the security guarantees of SSL/TLS connections by making SSL/TLS implementations accept invalid certificates.

B. A Motivating Example

To demonstrate the basic principles of our approach, let us consider the following example: suppose A and B are two different programs with similar functionality and that `checkVer_A` and `checkVer_B` are the functions validating the version number of the input files used by A and B respectively, as shown in Figure 1. Both of these functions return

[1]*Nezha* [5] is a Chinese deity commonly depicted in a "three heads and six arms" form. His multi-headed form is analogous to our tool, which peers into different programs to pinpoint discrepancies.

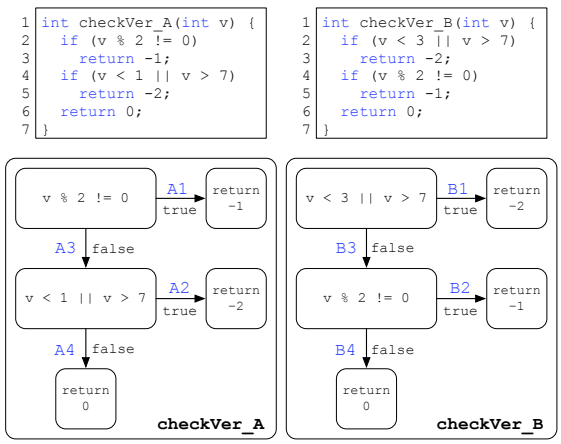

Fig. 1: (*Top*) Simplified example of a semantic discrepancy and (*Bottom*) the corresponding simplified Control Flow Graphs.

0 to indicate a valid version number or a negative number (-1 or -2) to indicate an error. While almost identical, the two programs have a subtle discrepancy in their validation behavior. In particular, `checkVer_A` accepts an input of `v=2` as valid while `checkVer_B` rejects it with an error code of `-2`.

The above example, albeit simplified, is similar to the semantic bugs found in deployed, real-world applications. This leads us to the following research question: *how can* NEZHA *efficiently generate test inputs that demonstrate discrepancies between similar programs?* Our key intuition is that simultaneously testing multiple programs on the same input offers a wide range of information that can be used to compare the tested programs' behaviors relative to each other. Such examples include error messages, debug logs, rendered outputs, return values, observed execution paths of each program, *etc.* Semantic discrepancies across programs are more likely for the inputs that cause relative variations of features like the above across multiple test programs. Adopting an evolutionary algorithm approach, NEZHA begins with a corpus of seed inputs, applies mutations to each input in the corpus, and then selects the best-performing inputs for further mutations. The fitness of a given input is determined based on the *diversity* it introduces in the observed behaviors across the tested programs. NEZHA builds upon this notion of differential diversity, utilizing two different *δ-diversity* guidance engines, one black-box and one-gray box.

1) Scenario 1: Gray-box Guidance: If program instrumentation is a feasible option, we can collect detailed runtime execution information from the test programs, for each input. For instance, knowledge of the portions of the Control Flow Graph (CFG) that are accessed during each program execution, can guide us into only mutating the inputs that are likely to visit new edges in the CFG. An edge in a CFG exists between two basic blocks if control may flow from one basic block to the other (e.g., A_1 is an edge in the simplified CFG for `checkVer_A` as shown in Figure 1). We illustrate how this information can be collectively tracked across multiple programs revisiting the example of Figure 1.

Suppose that our initial corpus of test files (seed corpus) consists of three input files, with versions $7, 0$, and 1 ($I_0 = \{7, 0, 1\}$). We randomly extract one input from I_0 to start our testing: suppose the input with `v=7` is selected and then passed to both `checkVer_A` and `checkVer_B`. As shown in Table I, the execution paths for programs A and B (i.e., the sequence of unique edges accessed during the execution of each program) are $\{A_1\}$ and $\{B_3, B_2\}$ respectively. The number of edges covered in each program is thus 1 and 2 for A and B respectively, whereas the coverage achieved across both programs is $1 + 2 = 3$. One may drive the input generation process favoring the mutation of inputs that increase coverage (i.e., exercise previously unexplored edges). Since `v=7` increased the code coverage, it is added to the corpus that will be used for the next generation: $I_1 = \{7\}$. In the following stage of the testing, we pick any remaining inputs from the current corpus and pass them to programs A and B. Selecting `v=0` as the next input will also increase coverage, since execution touches three previously-unseen edges (A_3, A_2 and B_1), and thus the file is picked for further mutations: $I_1 = \{7, 0\}$. At this stage, the only input of I_0 that has not been executed is `v=1`. This input's execution does not increase coverage, since both edges A_1 and B_1 have been visited again, and thus `v=1` is not added to I_1 and will not be considered for future mutations. However, we notice that `v=1`, with a single increment mutation, could be transformed to an input that would disclose the discrepancy between programs A and B, had it not been discarded. This example demonstrates that simply maximizing edge-coverage often *misses* interesting inputs that may trigger semantic bugs. By contrast, had we tracked the δ-diversity using path tuples across past iterations, input `v=1` would invoke the path tuple $\langle\{A_1\}, \{B_1\}\rangle$, which, as a *pair/combination*, would have not been seen before. Thus, using a *path δ-diversity* state, instead of code coverage, results in `v=1` been considered for further mutations. As seen in Table I, the mutated input `v=2` uncovers the semantic bug.

2) Scenario 2: Black-box Guidance: If program instrumentation or binary rewriting are not feasible options, we may still adapt the notion of program diversity to a black-box setting. The key intuition is, again, to look for previously unseen *patterns* across the observed outputs of the tested programs. Depending on the context of the application being tested, available outputs may vary greatly. For instance, a malware detector may only provide one bit of information based on whether some input file contains a malware or not, whereas other applications may offer richer sets of outputs such as graphical content, error or debug messages, values returned to the executing shell, exceptions, *etc.* In the context of differential testing, the outputs of a single application A can be used as a reference against the outputs of all other applications being tested. For example, if browsers A, B, and C are differentially tested, one may use browser A as a reference and then examine the contents of different portions of the

Generation	Mutation	Input	Execution Paths		Path Tuple	δ-diversity State	Add to Corpus		Report Bug	
			A	B			Coverage	δ-diversity	Coverage	δ-diversity
seed	-	7	$\{A_1\}$	$\{B_3, B_2\}$	$P_1 = \langle\{A_1\}, \{B_3, B_2\}\rangle$	$\{P_1\}$	✓	✓	✗	✗
seed	-	0	$\{A_3, A_2\}$	$\{B_1\}$	$P_2 = \langle\{A_3, A_2\}, \{B_1\}\rangle$	$\{P_1, P_2\}$	✓	✓	✗	✗
seed	-	1	$\{A_1\}$	$\{B_1\}$	$P_3 = \langle\{A_1\}, \{B_1\}\rangle$	$\{P_1, P_2, P_3\}$	✗	✓	✗	✗
1	increment	2	$\{A_3, A_4\}$	$\{B_1\}$	$P_4 = \langle\{A_3, A_4\}, \{B_1\}\rangle$	$\{P_1, P_2, P_3, P_4\}$	-	✓	-	✓

TABLE I: A semantic bug that is missed by differential testing using code coverage but can be detected by NEZHA's path δ-diversity (gray-box) during testing of the examples shown in Figure 1. NEZHA's black-box δ-diversity input generation scheme (not shown in this example) would also have found the semantic bug.

rendered Web pages with respect to A, using an arbitrary number of values for the encoding (different values may denote a mismatch in the CSS or HTML rendering *etc.*).

Regardless of the output formulation, however, for each input used during testing, NEZHA may receive a corresponding set of *output values* and then only select the inputs that result in new output tuples for further mutations. In the context of the example of Figure 1, let us assume that the outputs passed to NEZHA are the values returned by routines `checkVer_A` and `checkVer_B`. If inputs $0, 7$, and 1 are passed to programs A and B, NEZHA will update its internal state with all unique output tuples seen so far: $\{\langle -1, -1\rangle, \langle -2, -2\rangle, \langle -1, -2\rangle\}$. Any new input which will result in a previously unseen tuple will be considered for future mutations, otherwise it will be discarded (e.g., with the aforementioned output tuple set, input 2 resulting in tuple $\langle 0, -2\rangle$ would be considered for future mutations, but input 9 resulting in $\langle -1, -2\rangle$ would be discarded).

III. METHODOLOGY

In each testing session, NEZHA observes the relative behavioral differences across all tested programs to maximize the number of reported semantic bugs. To do so, NEZHA uses Evolutionary Testing (ET) [53], inferring correlations between the inputs passed to the tested applications and their observed behavioral asymmetries, and, subsequently, refines the input generation, favoring more promising inputs. Contrary to existing differential testing schemes that drive their input generation using *monolithic* metrics such as the code coverage that is maximized across some or all of the tested programs, NEZHA utilizes the novel concept of δ-diversity: metrics that preserve the *differential diversity* (δ-diversity) of the tested applications will perform better at finding semantic bugs than metrics that overlook relative asymmetries in the applications' execution. The motivation behind δ-diversity becomes clearer if we examine the following example. Suppose we are performing differential testing between applications A and B. Now, suppose an input I_1 results in a combined coverage C across A and B, exercising 30% of the CFG edges in A and 10% of the edges in B. A different input I_2, that results in the same overall coverage C, however exercising 10% of the edges in A and 28% of the edges of B, would not be explored further under monolithic schemes, despite the fact that it exhibits much different behavior in each application compared to input I_1.

Algorithm 1 DiffTest: Report all discrepancies across applications $\mathcal{A}$ after n generations, starting from a corpus $\mathcal{I}$

```
1:  procedure DIFFTEST(I, A, n, GlobalState)
2:      discrepancies = ∅ ;reported discrepancies
3:      while generation ≤ n do
4:          input = RANDOMCHOICE(I)
5:          mut_input = MUTATE(input)
6:          generation_paths = ∅
7:          generation_outputs = ∅
8:          for app ∈ A do
9:              app_path, app_outputs = RUN(app, mut_input)
10:             genereration_paths ∪= {app_path}
11:             genereration_outputs ∪= {app_outputs}
12:         end for
13:         if NEWPATTERN(generation_paths,
                          generation_outputs,
                          GlobalState) then
14:             I ← I ∪ mut_input
15:         end if
16:         if ISDISCREPANCY(generation_outputs) then
17:             discrepancies ∪= mut_input
18:         end if
19:         generation = generation + 1
20:     end while
21:     return discrepancies
22: end procedure
```

We present NEZHA's core engine in Algorithm 1. In each testing session, NEZHA examines if different inputs result in previously unseen relative execution *patterns* across the tested programs. NEZHA starts from a set of initial seed inputs $\mathcal{I}$, and performs testing on a set of programs $\mathcal{A}$ for a fixed number of generations (n). In each generation, NEZHA randomly selects (line 4) and mutates (line 5) one input (individual) out of the population $\mathcal{I}$, and tests it against each of the programs in $\mathcal{A}$. The recorded execution paths and outputs for each application are added to the sets of total paths and outputs observed during the current generation (lines 8-12). Subsequently, if NEZHA determines that a *new execution pattern* is observed during this input execution, it adds the respective input to the input corpus, which will be used to produce the upcoming generation (lines 13-14). Finally, if there is a discrepancy in the outputs of the tested applications, NEZHA adds the respective input to the set of total discrepancies found (lines 16-18). Whether a discrepancy is observed in each generation depends on the outputs of the tested programs: if at least one application rejects an input and at least one other accepts it, a discrepancy

is logged.

A. Guidance Engines

In Algorithm 1, we demonstrated that NEZHA adds an input to the active corpus only if that input exhibits a newly seen pattern. In traditional evolutionary algorithms, the fitness of an individual for producing future generations is determined by its fitness score. In this section, we explain how δ-diversity can be used in NEZHA's guidance engines, both in a gray-box and a black-box setting.

1) Gray-box guidance: The most prevalent guidance mechanism in gray-box testing frameworks is the code coverage achieved by individual inputs across the sets of tested applications. Code coverage can be measured using function coverage (i.e., the functions accessed in one execution run), basic block coverage or edge coverage. However, as discussed in Section II, this technique is not well suited for finding semantic bugs. By contrast, NEZHA leverages relative asymmetries of the executed program paths to introduce two novel δ-diversity *path selection* guidance engines, suitable for efficient differential testing.

Suppose a program p is executing under an input i. We call the sequence of edges accessed during this execution the *execution path* of p under i, denoted by $path_{p,i}$. Tracking all executed paths (i.e., all the sequences of edges accessed in the CFG) is impractical for large-scale applications containing multiple loops and complex function invocations. In order to avoid this explosion in tracked states, NEZHA's gray-box guidance uses two different approximations of the execution paths, one of coarse granularity and the other offering finer tracking of the relative execution paths.

Path δ-diversity (coarse): Given a set of programs $\mathcal{P}$ that are executing under an input i, let $PC_{\mathcal{P},i}$ be the *Path Cardinality* tuple $\langle |path_{p_1,i}|, |path_{p_2,i}|, ..., |path_{p_{|\mathcal{P}|},i}| \rangle$. Each $PC_{\mathcal{P},i}$ entry represents the total number of edges accessed in each program $p_k \in \mathcal{P}$, for one *single* input i. Notice that $PC_{\mathcal{P},i}$ differs from the total coverage achieved in the execution of programs $\mathcal{P}$ under i, in the sense that $PC_{\mathcal{P},i}$ does not maintain a global, monolithic score, but a per-application count of the edges accessed, when each program is executing under input i. Throughout an entire testing session, starting from an initial input corpus $\mathcal{I}$, the overall (coarse) path δ-diversity achieved is the cardinality of the set containing all the above tuples: $PD_{Coarse} = |\bigcup_{i \in \mathcal{I}} \{PC_{\mathcal{P},i}\}|$.

This representation expresses the maximum number of unique path cardinality *tuples* for all programs in $\mathcal{P}$ that have been seen throughout the session. However, we notice that, although the above formulation offers a semantically richer representation of the execution, compared to total edge coverage, it constitutes a coarse approximation of the (real) execution paths. A finer-grained representation of the execution can be achieved if we take into account *which* edges, specifically, have been accessed.

Path δ-diversity (fine): Consider the path $path_{p,i}$, which holds all edges accessed during an execution of each program $p_k \in \mathcal{P}$ under input i. Let $path\_set_{p,i}$ be the set consisting of all unique edges of $path_{p,i}$. Thus $path\_set_{p,i}$ contains no duplicate edges, but instead holds only the CFG edges of p that have been accessed *at least once* during the execution. Given a set of programs $\mathcal{P}$, the (fine) path diversity of input i across $\mathcal{P}$ is the tuple $PD_{\mathcal{P},i} = \langle path\_set_{p_1,i}, path\_set_{p_2,i}, ..., path\_set_{p_{|\mathcal{P}|},i} \rangle$. Essentially, $PD_{\mathcal{P},i}$ acts as a "fingerprint" of the execution of input i across all tested programs and encapsulates relative differences in the execution paths *across* applications. For an entire testing session, starting from an initial input corpus $\mathcal{I}$, the (fine) path δ-diversity achieved is the cardinality of the set containing all the above tuples: $PD_{Fine} = |\bigcup_{i \in \mathcal{I}} \{PD_{\mathcal{P},i}\}|$.

To demonstrate how the above metrics can lead to different discrepancies, let us consider a differential testing session involving two programs A and B. Let A_n, B_n denote edges in the CFG of A and B, respectively, and let us assume that a given test input causes the paths $\langle A_1, A_2, A_1 \rangle$ and $\langle B_1 \rangle$ to be exercised in A and B respectively. At this point, $PD_{Coarse} = \{\langle 3, 1 \rangle\}$, and $PD_{Fine} = \{\langle \{A_1, A_2\}, \{B_1\} \rangle\}$. Suppose we mutate the current input, and the second (mutated) input now exercises paths $\langle A_1, A_2 \rangle$ and $\langle B_1 \rangle$ across the two applications. After the execution of this second input, PD_{Fine} remains unchanged, because the tuple $\langle \{A_1, A_2\}, \{B_1\} \rangle$ is already in the PD_{Fine} set. Conversely, PD_{Coarse} will be updated to $PD_{Coarse} = \{\langle 3, 1 \rangle, \langle 2, 1 \rangle\}$. Therefore, the new input will be considered for further mutation under a coarse path guidance, since it increased the cardinality of the PD_{Coarse} set, however it will be rejected under fine δ-diversity guidance. Finally, note that if we use total edge coverage as our metric for input selection, both the first and second inputs result in the same code coverage of 3 edges (two unique edges for A plus one edge for B). Thus, under a coverage-guided engine, the second input will be rejected as it does not increase code coverage, *despite* the fact that it executes in a manner that has not been previously observed across the two applications.

2) Black-box guidance: As mentioned in Section II-B2, NEZHA's input generation can be driven in a black-box manner using any observable and countable program output, such as error/debug messages, rendered or parsed outputs, return values *etc.* For many applications, especially those implementing particular protocols or RFCs, such outputs often uniquely identify deterministic execution patterns. For example, when a family of similar programs returns different error codes/messages, any change in one test program's returned error relative to the error codes returned by the other programs is highly indicative of the relative behavioral differences between them. Such output asymmetries can be used to guide NEZHA's path selection.

Output δ-diversity: Let p be a program which, given an input i, produces an output $o_{p,i}$. We define the output diversity of a family of programs $\mathcal{P}$, executing with a single input i, as the tuple $OD_{\mathcal{P},i} = \langle o_{p_1,i}, o_{p_2,i}, ..., o_{p_{|\mathcal{P}|},i} \rangle$. Across a testing session that starts from an input corpus $\mathcal{I}$, output δ-diversity tracks the number of unique output tuples that are observed throughout the execution of inputs $i \in \mathcal{I}$ across all programs in $\mathcal{P}$: $|\bigcup_{i \in \mathcal{I}} \{OD_{\mathcal{P},i}\}|$. Input generation based

on output δ-diversity aims to drive the tested applications to result in as many different output combinations across the overall pool of programs, as possible. This metric requires no knowledge about the internals of each application and is completely black-box. As a result, it can even be applied on applications running on a remote server or in cases were binary rewriting or instrumentation is infeasible. We demonstrate in Section V that this metric performs equally well as NEZHA's gray-box engines for programs that support fine-grained error values.

Algorithm 2 Determine if a new pattern has been observed

```
1:  procedure NEWPATTERN(gen_paths,
                         gen_outputs,
                         GlobalState)
2:      IsNew =false
3:      if GlobalState.UsePDCoarse then
4:          IsNew | = PDCOARSE(gen_paths, GlobalState)
5:      end if
6:      if GlobalState.UsePDFine then
7:          IsNew | = PDFINE(gen_paths, GlobalState)
8:      end if
9:      if GlobalState.UseOD then
10:         IsNew | = OD(gen_outputs, GlobalState)
11:     end if
12:     return IsNew
13: end procedure
```

As described in Algorithm 1, whenever a set of applications is tested under NEZHA, a mutated input that results in a previously unseen pattern (Algorithm 1 - lines 13-15) is added to the active input corpus to be used in future mutations. Procedure `NewPattern` is called for each input (at every generation), after all tested applications have executed, to determine if the input exhibits a newly observed behavior and should be added in the current corpus. The pseudocode for the routine is described in Algorithm 2: for each of the active guidance engines in use, NEZHA calls the respective routine listed in Algorithm 3 and, if the path δ-diversity and output δ-diversity is increased for each of the modes respectively (i.e., the input results in a discovery of a previously unseen tuple), the mutated input is added to the current corpus.

B. Automated Debugging

NEZHA is designed to efficiently detect discrepancies across similar programs. However, the larger the number of reported discrepancies and the larger the number of tested applications, the harder it is to identify unique discrepancies and to localize the root cause of each report. To aid bug localization, NEZHA stores each mutated input in its original form throughout the execution of each generation. NEZHA compares any input that caused a discrepancy with its corresponding stored copy (before the mutation occurred), and logs the difference between the two. As this input pair differs only on the part that introduced the discrepancy, the two inputs can subsequently be used for delta-debugging [67] to pinpoint the root cause of the difference. Finally, to aid *manual analysis* of reported discrepancies, NEZHA performs a bucketing of reported differences using the return values of the tested programs. Moreover, it reports the file similarity of reported discrepancies using context-triggered piece-wise fuzzy hashing [45]. Automated debugging and bug localization in the context of differential testing is not trivial. Future additions in the current NEZHA design, as well as limitations of existing techniques are discussed further in Section VIII.

Algorithm 3 NEZHA path selection routines

```
1:  ; Path δ-diversity (coarse)
2:  ; @generation_paths: paths for each tested app for current input
3:  ; @GS: GlobalState (bookkeeping of paths, scores etc.)
4:  procedure PDCOARSE(generation_paths, GS)
5:      path_card = ∅
6:      for path in generation_paths do
7:          path_card ∪ = {|path|}
8:      end for
9:      ; See if the path_card tuple has been seen before:
10:     ; check against stored tuples in the GlobalState
11:     new_card_tuple = {⟨path_card⟩} \ GS.PDC_tuples
12:     if new_card_tuple ≠ ∅ then
13:         ; If new, add to GlobalState and update score
14:         GS.PDC_tuples ∪ = new_card_tuple
15:         GlobalState.PDC_Score = |GS.PDC_tuples|
16:         return true
17:     end if
18:     return false
19: end procedure

20: ; Path δ-diversity (fine)
21: procedure PDFINE(generation_paths, GS)
22:     path_set = ∅
23:     for path in generation_paths do
24:         path_set ∪ = {path}
25:     end for
26:     new_paths = {⟨path_set⟩} \ GS.PDF_tuples
27:     if new_path_tuple ≠ ∅ then
28:         GS.PDF_tuples ∪ = new_path_tuple
29:         GlobalState.PDF_Score = |GS.PDF_tuples|
30:         return true
31:     end if
32:     return false
33: end procedure

34: ; Output δ-diversity
35: procedure OD(generation_outputs, GS)
36:     new_output_tuple = {⟨output_tuple⟩} \ GS.OD_tuples
37:     if new_output_tuple ≠ ∅ then
38:         GS.OD_tuples ∪ = new_output_tuple
39:         GlobalState.OD_Score = |GS.OD_tuples|
40:         return true
41:     end if
42:     return false
43: end procedure
```

IV. SYSTEM DESIGN AND IMPLEMENTATION

A. Architecture Overview

We present NEZHA's architecture in Figure 2. NEZHA consists of two main components: its core engine and runtime components. The runtime component collects all information necessary for NEZHA's δ-diversity guidance and subsequently

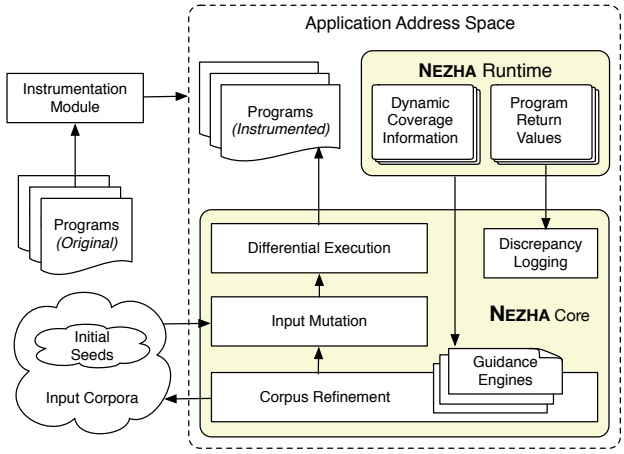

Fig. 2: System architecture.

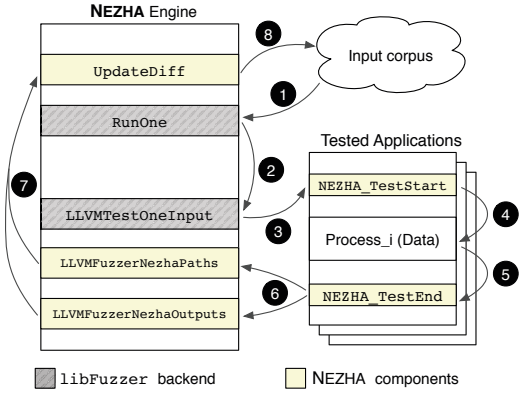

Fig. 3: Example of how an input is processed through NEZHA.

passes it to the core engine. The core engine then generates new inputs through mutations, and updates the input corpus based on its δ-diversity guidance.

We implemented NEZHA using Clang v3.8. Our implementation consists of a total of 1545 lines of C++ code, of which 1145 and 400 lines correspond to NEZHA's core and runtime components, respectively.

B. Instrumentation

To enable NEZHA's gray-box guidance, the test programs must be instrumented to gather information on the paths executed for each test input. This can be achieved either during compilation, using dynamic binary instrumentation, or using binary rewriting. For our prototype, we instrument programs at compile-time, using Clang's SanitizerCoverage [6]. SanitizerCoverage can be combined with one or more of Clang's sanitizers, namely AddressSanitizer (ASAN) [57], UndefinedBehaviorSanitizer (UBSAN) [8], and MemorySanitizer (MSAN) [60], to achieve memory error detection during testing. In our implementation, we instrument the test programs with Clang's ASAN to reap the benefit of finding potential memory corruption bugs in addition to discrepancies with a nominal overhead. We note that ASAN is not strictly required for us to find discrepancies in our experiments.

C. NEZHA Core Engine and Runtime

NEZHA's core engine is responsible for driving the input generation process using the guidance engines described in Section III-A. We implement the core NEZHA engine by adapting and modifying libFuzzer [4], a popular coverage-guided evolutionary fuzzer that has been successful in finding large numbers of non-semantic bugs in numerous large-scale, real-world software. libFuzzer primarily focuses on library fuzzing, however it can be adapted to fuzz whole applications, passing the path and output information needed to guide the generation of inputs as parameters to the main engine. NEZHA's δ-diversity engine is *independent of the underlying testing framework*, and can be applied as-is to *any* existing fuzzer or differential testing engine, whether black-box or white-box/gray-box. Our choice of extending libFuzzer is due to its large adoption, as well as its modularity, which allows for a real-world evaluation of NEZHA's δ-diversity with a state-of-the-art code coverage-based framework.

LibFuzzer provides API support for custom input mutations, however it is not designed for differential testing nor does it support modifications of its internal structures. With respect to mutations, we do not customize libFuzzer's engine so that we can achieve a fair comparison of NEZHA's δ-diversity with the default coverage-based guidance of the fuzzer, keeping all other components intact. Instead, NEZHA uses libFuzzer's built-in engine to apply up to a maximum of five of the following mutation operators in random order: i) create a new input by combining random substrings from different inputs, ii) add/remove an existing byte from an input, iii) randomize a bit/byte in the input, iv) randomly change the order of a subset of the input bytes and, v) only randomize the bytes whose value corresponds to the ASCII code of a digit character (i.e., 0x30-0x39). Finally, besides adding support for NEZHA's δ-diversity to libFuzzer, we also extend its guidance engines to support (global) code coverage guidance in the context of differential testing. As we will demonstrate in Section V, δ-diversity outperforms code coverage, even when the latter is applied across all tested applications.

A NEZHA-instrumented program can be executed using any of NEZHA's guidance engines, as long as the binary is invoked with the appropriate runtime flags. In libFuzzer, customized test program invocation is achieved overriding the `LLVMFuzzerTestOneInput` function. We override this function to load NEZHA into a main driver program, which then performs the differential testing across all examined applications. We also extend libFuzzer with two additional API calls, `LLVMFuzzerNezhaOutputs` and `LLVMFuzzerNezhaPaths` that provide interfaces for passing output values and path execution information between the core NEZHA engine and the NEZHA library running as part of the tested programs. Finally, the NEZHA runtime uses two API calls, namely `NEZHA_TestStart` and `NEZHA_TestEnd`,

that the core engine can use to perform per-program appropriate initialization and cleanup operations respectively (allocation and deallocation of buffers holding path and output information throughout the execution *etc.*).

In Figure 3, we present an example of how an input is used by NEZHA and how the various components interoperate. Assume that the NEZHA engine begins by selecting an input from the corpus at Step ❶. It then mutates the input and dispatches it to the tested programs through `LLVMFuzzerTestOneInput` at Step ❷. At Step ❸, the NEZHA library initializes all its bookkeeping data structures for each of the invoked applications via the `NEZHA_TestStart` call, and subsequently invokes the program-specific functionality under test at Step ❹. Upon completion, NEZHA deinitializes temporary bookkeeping data at Step ❺. The runtime execution information is dispatched back to the NEZHA engine through the designated API invocations at Step ❻. Finally, at Step ❼, the δ-diversity engine in use determines if the input will be added to the corpus for further testing. If so, the input is added to the corpus at Step ❽.

V. EVALUATION

In this section, we assess the effectiveness of NEZHA both in terms of finding discrepancies in security-critical, real-world software, as well as in terms of its core engine's efficiency compared to other differential testing tools. In particular, we evaluate NEZHA by differentially testing six major SSL libraries, file format parsers, and PDF viewers. We also compare NEZHA against two domain-specific differential testing engines, namely Frankencerts [24] and Mucerts [32], and two state-of-the-art domain-agnostic guided mutational fuzzers: American Fuzzy Lop (AFL) [66], and libFuzzer [4]. Our evaluation aims at answering the following research questions: 1) is NEZHA effective at finding semantic bugs? 2) does it perform better than domain-specific testing engines? 3) does it perform better than domain-agnostic coverage-guided fuzzers? 4) what are the benefits and limitations of each of NEZHA's δ-diversity engines?

A. Experimental Setup

X.509 certificate validation: We examine six major SSL libraries, namely OpenSSL (v1.0.2h), LibreSSL (v2.4.0), BoringSSL (f0451ca[2]), wolfSSL (v3.9.6), mbedTLS (v2.2.1) and GnuTLS (v3.5.0). Each of the SSL/TLS libraries is instrumented with SanitizerCoverage and AdressSanitizer so that NEZHA has access to the programs' path and output information. For each library, NEZHA invokes its built-in certificate validation routines and compares the respective error codes: if at least one library returns an error code on a given certificate whereas another library accepts the same certificate, this is counted as a discrepancy.

For our experiments, our pool of seed inputs consists of 205,853 DER certificate chains scraped from the Web. Out of these, we sampled certificates to construct 100 distinct groups of 1000 certificates each. Initially, *no certificate in any of the initial 100 groups introduced a discrepancy* between the tested applications thus all reported discrepancies in our results are introduced solely due to the differential testing of the examined frameworks.

ELF and XZ parsing: We evaluate NEZHA on parsers of two popular file formats, namely the ELF and the XZ formats. For parsing of ELF files, we compare the parsing implementations in the ClamAV malware detector with that of the `binutils` package, which is ubiquitous across Unix/Linux systems. In each testing session, NEZHA loads a file and validates it using ClamAV and `binutils` (the respective validation libraries are `libclamav` and `libbfd`), and either reports it as a valid ELF binary or returns an appropriate error code. Both programs, including all their exported libraries, are instrumented to work with NEZHA and are differentially tested for a total of 10 million generations. In our experiments, we use ClamAV 0.99.2 and `binutils` v.2.26-1-1_all. Our seed corpus consists of 1000 Unix malware files sampled from VirusShare [9] and a plain '*hello world*' program.

Similar to the setup for ELF parsing, we compare the XZ parsing logic of ClamAV and XZ Utils [19], the default Linux/Unix command-line decompression tool for XZ archive files. The respective versions of the tested programs are ClamAV 0.99.2 and `xzutils` v5.2.2. Our XZ seed corpus uses the XZ files from the XZ Utils test suite (a total of 74 archives) and both applications are differentially tested for a total of 10 million generations.

PDF Viewers: We evaluate NEZHA on three popular PDF viewers, namely the Evince (v3.22.1), MuPDF (v1.9a) and Xpdf (v3.04) viewers. Our pool of tested inputs consists of the PDFs included in the Isartor [3] testsuite. All applications are differentially tested for a total of 10 million generations. During testing, NEZHA forks a new process for each tested program, invokes the respective binary through `execlp`, and uses the return values returned by the execution to the parent process to guide the input generation using its output δ-diversity. Determined based on the return values of the tested programs, the discrepancies constitute a conservative estimate of the total discrepancies, because while the return values of the respective programs may match, the rendered PDFs may differ.

All our measurements were performed on a system running Debian GNU/Linux 4.5.5-1 while our implementation of NEZHA was tested using Clang version 3.8.

Q1: How effective is NEZHA *in discovering discrepancies?*

The results of our analysis with respect to the discrepancies and memory errors found are summarized in Table II. NEZHA found 778 validation discrepancies and 8 memory errors in total. Each of the reported discrepancies corresponds to a unique tuple of error codes, where at least one application accepts an input and at least another application rejects it. Examples of semantic bugs found are presented in Section VI.

[2]This refers to a git commit hash from BoringSSL's master branch

Type	SSL Certificate	XZ Archive	ELF Binary	PDF File
Discrepancies	764	5	2	7
Errors & Crashes	6	2	0	0

TABLE II: Result summary for our analysis of NEZHA.

We observe that, out of the total 778 discrepancies, 764 were reported during our evaluation of the tested SSL/TLS libraries. The disproportionately large number of discrepancies found for SSL/TLS is attributed to the fine granularity of the error codes returned by these libraries, as well as to the larger number of applications being tested (six applications for SSL/TLS versus three for PDF and two for ELF/XZ).

To provide an insight into the impact that the number of tested programs has over the total reported discrepancies, we measure the total discrepancies observed between every *pair* of the six SSL/TLS libraries. In the pair-wise comparison of Table III, two different return-value tuples that have the same error codes for libraries A and B are not counted twice for the (A, B) pair (i.e., we regard the output tuples $\langle 0, 1, 2, 2, 2, 2\rangle$ and $\langle 0, 1, 3, 3, 3, 3\rangle$ as one pairwise discrepancy with respect to the first two libraries). We observe that *even in cases of very similar code bases* (e.g., OpenSSL and LibreSSL which are forks of the same code base), NEZHA *successfully reports multiple unique discrepancies*.

	LibreSSL	BoringSSL	wolfSSL	mbedTLS	GnuTLS
OpenSSL	10	1	8	33	25
LibreSSL	-	11	8	19	19
BoringSSL	-	-	8	33	25
wolfSSL	-	-	-	6	8
mbedTLS	-	-	-	-	31

TABLE III: Number of *unique pairwise* discrepancies between different SSL libraries. Note that the input generation is still guided using *all* of the tested SSL/TLS libraries.

The results presented in Table II are new reports and not reproductions of existing ones. They include multiple **confirmed, previously unknown semantic errors**. Moreover, NEZHA was more efficient at reporting discrepancies than all guided or unguided frameworks we compared it against (see Q2 & Q3 for further details on this analysis). We present some examples of semantic bugs that have already been *identified and patched* by the respective software development teams in Section VI.

Result 1: NEZHA reported 778 previously unknown discrepancies (including confirmed security vulnerabilities and semantic errors), in total, across all the applications we tested, even when the latter shared similar code bases.

In addition to finding semantic bugs, NEZHA was equally successful in uncovering previously unknown memory corruption vulnerabilities and crashes in the tested applications. In particular, five of them were crashes due to invalid memory accesses (four cases in wolfSSL and one in GnuTLS), one was a memory leak in GnuTLS and two were use-after-free bugs in ClamAV. As NEZHA's primary goal is to find semantic bugs (not memory corruption issues), we do not describe them in detail here. Interested readers can find further details in Section XI-A of the Appendix.

Q2: How does NEZHA *perform compared to domain-specific differential testing frameworks like Frankencerts and Mucerts?*

One may argue that being domain-independent, NEZHA may not be as efficient as successful domain-specific frameworks. To address this concern, we compared NEZHA against Frankencerts [24], a popular black-box unguided differential testing framework for SSL/TLS certificate validation, as well as Mucerts [32], which builds on top of Frankencerts performing Markov Chain Monte Carlo (MCMC) sampling to diversify certificates using coverage information. Frankencerts generates mutated certificates by randomly combining X.509 certificate fields that are decomposed from a corpus of seed certificates. Despite its unguided nature, Frankencerts successfully uncovered a multitude of bugs in various SSL/TLS libraries. Mucerts adapt many of Frankencerts core components but also stochastically optimize the certificate generation process based on the coverage each input achieves in a single application (OpenSSL). Once the certificates have been generated from this single program, they are used as inputs to differentially test all SSL/TLS libraries.

To make a fair comparison between NEZHA, Frankencerts, and Mucerts, we ensure that all tools are given the same sets of input seeds. Furthermore, since Frankencerts is a black-box tool, we restrict NEZHA to only use its black-box output δ-diversity guidance, across all experiments.

Since the input generation is stochastic in nature due to the random mutations, we perform our experiments with multiple runs to obtain statistically sound results. In particular, for each of the input groups of certificates we created (100 groups of 1000 certificates each), we generate $100,000$ certificate chains using Frankencerts, resulting in a total of 10 million Frankencerts-generated chains. Likewise, passing as input each of the above 100 corpuses, we run NEZHA for $100,000$ generations (resulting in 10 million NEZHA-executed inputs). Mucerts also start from the same sets of inputs and execute in mode 2, which according to [32] yields the most discrepancies with highest precision. We use the return value tuples of the respective programs to identify unique discrepancies (i.e., unique tuples of return values seen during testing).

We present the relative number and distribution of discrepancies found across Frankencerts, Mucerts and NEZHA in Figures 4 and 5. Overall, NEZHA reported 521 unique discrepancies, compared to 10 and 19 distinct discrepancies for Frankencerts and Mucerts respectively. NEZHA reports 52 times and 27 times more discrepancies than Frankencerts and Mucerts respectively, *starting from the same sets of initial seeds* and running for the same number of iterations, achieving a respective coverage increase of 15.22% and 33.48%.

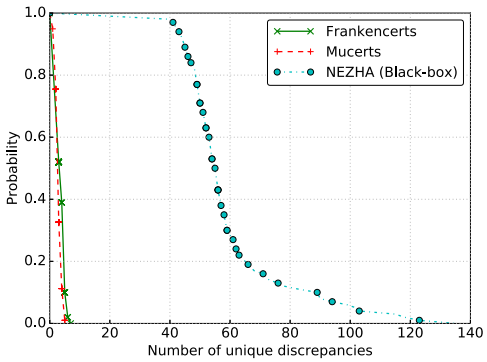

Fig. 4: Probability of finding at least n unique discrepancies starting from the same seed corpus of 1000 certificates and running $100,000$ iterations. The results are averages of 100 runs each starting with a different seed corpus.

We observe that, while both Frankencerts and Mucerts reported a much smaller number of discrepancies than NEZHA, they found 3 and 15 discrepancies respectively that were missed by NEZHA. We posit that this is due to the differences in their respective mutation engines. Frankencerts and Mucerts start from a corpus of certificates, break all the certificates in the corpus into the appropriate fields (extensions, dates, issuer *etc.*), then randomly sample and mutate those fields to merge them back together in *new* chains, however respecting the semantics of each field (for instance, Frankencerts might mutate and merge the extensions of two or three certificates to form the extensions field of a new chain but will not substitute a date field with an extension field). On the contrary, NEZHA performs its mutations sequentially, without mixing together different components of the certificates in the seed corpus, as it does not have any knowledge of the input format.

It is noteworthy that, despite the fact that NEZHA's mutation operators are domain-independent, NEZHA's guidance mechanism allows it to favor inputs that are *mostly syntactically correct*. Compared to Frankencerts or Mucerts that mutate certificates at the granularity of X.509 certificate fields, without violating the core structure of a certificate, NEZHA still yields more bugs. Finally, when running NEZHA's mutation engine without any guidance, on the same inputs, we observe that no discrepancies were found. Therefore, NEZHA's efficacy in finding discrepancies can only be attributed to its black-box δ-diversity-based guidance.

Result 2: NEZHA reports 52 times and 27 times more discrepancies than Frankencerts and Mucerts respectively, per input. In terms of testing performance, NEZHA analyzes more than 400 certificates per second, compared to 271 and 0.08 certificates per second for Frankencerts and Mucerts respectively.

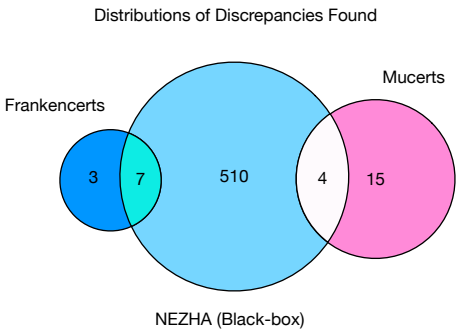

Fig. 5: Unique discrepancies observed by Frankencerts, Mucerts and NEZHA (black-box). The results are averages of 100 runs each starting with a different seed corpus of 1000 certificates.

Q3: How does NEZHA *perform compared to state-of-the art coverage-guided domain-independent fuzzers like AFL/libfuzzer?*

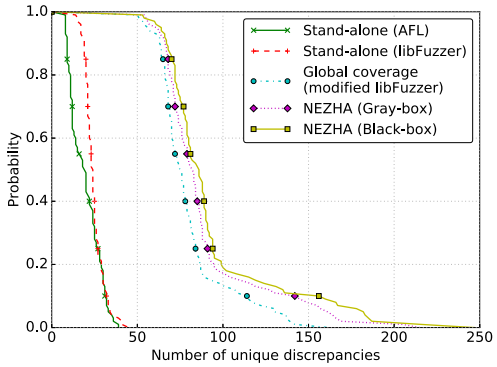

Fig. 6: Probability of finding at least n unique discrepancies after $100,000$ executions, starting from a corpus of 1000 certificates. The results are averages of 100 runs each starting from a different seed corpus of 1000 certificates.

None of the state-of-the-art domain-agnostic fuzzers like AFL natively support differential testing. However, they can be adapted for differential testing by using them to generate inputs with a single test application and then invoking the full set of tested applications with the generated inputs. To differentially test our suite of six SSL/TLS libraries, we first generate certificates using a coverage-guided fuzzer on OpenSSL, and then pass these certificates to the rest of the SSL libraries, similar to how differential testing is performed by Mucerts. The discrepancies reported across all tested SSL libraries, if we run AFL (v. 2.35b)[3] and libFuzzer on a standalone program (OpenSSL) are reported in Figure 6. We notice

[3]Since version 2.33b, AFL implements the `explore` schedule as presented in AFLFast [23], thus we omit comparison with the latter.

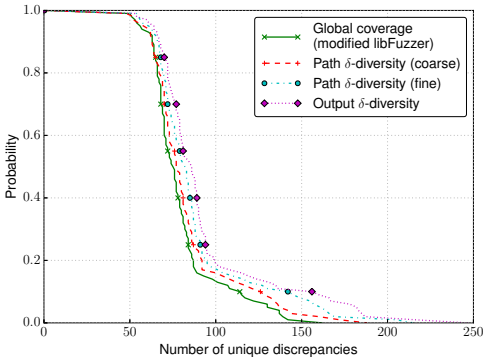

Fig. 7: Probability of finding at least n unique discrepancies for each of NEZHA's δ-diversity engines after $100,000$ executions. The results are averages of 100 runs each starting from a different seed corpus of 1000 certificates.

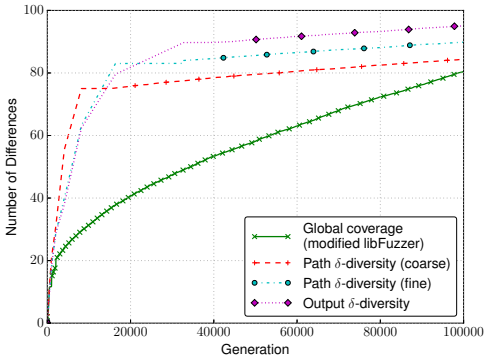

Fig. 8: Unique discrepancies observed for each of NEZHA's δ-diversity engines per generation. The results are averages of 100 runs each starting from a different seed corpus of 1000 certificates.

that NEZHA yields 6 times and 3.5 times more differences per tested input, on average, than AFL and libFuzzer respectively.

This demonstrates that driving input generation with a single application is ill-suited for differential testing. In the absence of a widely-adopted domain-agnostic differential testing framework, we *modified* libFuzzer's guidance engine to support differential testing using *global* code coverage. Apart from its guidance mechanisms, this modified libFuzzer [4] is identical to NEZHA in terms of all other aspects of the engine (mutations, corpus minimization *etc.*). Even so, as shown in Figure 6, NEZHA still yields 30% more discrepancies per tested input. Furthermore, NEZHA also achieves 1.3% more code coverage.

Result 3: NEZHA finds 6 times more discrepancies than AFL adapted to differentially test multiple applications using a single test program for input generation.

Q4: How does the performance of NEZHA*'s δ-diversity black-box and gray-box engines compare to each other?*

To compare the performance of NEZHA's δ-diversity engines, we run NEZHA on the six SSL/TLS libraries used in our previous experiments, enabling a single guidance engine at a time. Before evaluating NEZHA's δ-diversity guidance, we ensured that the discrepancies reported are a result of NEZHA's guidance and not attributed to NEZHA's mutations. Indeed, when we use NEZHA without any δ-diversity guidance, *no* discrepancies were found across the SSL/TLS libraries.

Figures 7 and 8 show the relative performances of different δ-diversity engines in terms of the number of unique discrepancies they discovered. Figure 7 shows the probability of finding at least n unique discrepancies across the six tested SSL/TLS libraries, starting from a corpus of 1000 certificates and performing $100,000$ generations. For this experimental setting, we notice that NEZHA reports at least 57 discrepancies with more than 90% probability regardless of the engine used. Furthermore, all δ-diversity engines report more discrepancies than global coverage. Figure 8 shows the rate at which each engine finds discrepancies during execution. We observe that both δ-diversity guidance engines report differences at higher rates than global coverage using the same initial set of inputs.

Overall throughout this experiment, NEZHA's output δ-diversity yielded 521 discrepancies, while path δ-diversity yielded 491 discrepancies, resulting in 30% and 22.75% more discrepancies than using global code coverage to drive the input generation (global coverage resulted in 400 unique discrepancies). With respect to the coverage of the CFG that is achieved, output δ-diversity and path δ-diversity guidance achieves 1.38% and 1.21% higher coverage then global coverage guidance (graphs representing the coverage and population increase at each generation are presented in Section XI-B).

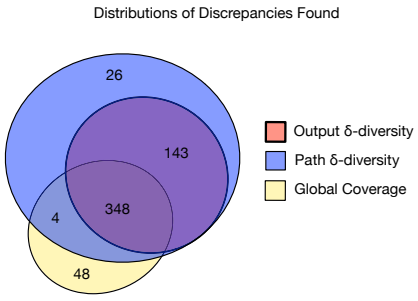

Fig. 9: Distribution of bugs found by NEZHA's δ-diversity engines versus NEZHA using global-coverage-based guidance.

The distribution of the discrepancies reported by the different engines is presented in Figure 9. We notice that 348 discrepancies have been found by all three guidance engines, 121 discrepancies are reported using δ-diversity and

[4]Corresponding git commit is 1f0a7ed0f324a2fb43f5ad2250fba68377076622

48 discrepancies are reported by our custom libFuzzer global code coverage engine. This result is a clear indication that δ-diversity performs differently than global code coverage with respect to input generation, generating a *broader* set of discrepancies for a *given time budget*, while exploring similar portions of the application CFG (1.21% difference in coverage for the same setup).

One notable result from this experiment is that output δ-diversity, despite being black-box, achieves equally good coverage with NEZHA's gray-box engines and even reports more unique discrepancies. This is a very promising result as it denotes that the internal state of an application can, in some cases, be adequately approximated based on its outputs alone assuming that there is enough diversity in the return values.

Result 4: NEZHA's output and path δ-diversity guidance finds 30% and 22.75% more discrepancies, respectively, than NEZHA using global-coverage-based guidance.

However, we expect that output δ-diversity will perform worse for applications for which the granularity of the outputs is very coarse. For instance, the discrepancies that will be found in an application that provides debug messages or fine-grained error codes are expected to be more than those found in applications with less expressive outputs, (e.g., a web application firewall that only returns ACCEPT or REJECT based on its input). To verify this assumption, we perform an experiment with only three SSL libraries, i.e., OpenSSL, LibreSSL and BoringSSL, in which all libraries are only returning a *subset* of their supported error codes, namely at most $32, 64, 128$ and 256 error codes. Our results are presented in Figure 10. We notice that a limit of 32 error codes results in significantly fewer discrepancies than a more expressive set of error values. Finally, we should note that when we decreased this limit further, to only allow 16 possible error codes across all three libraries, NEZHA did not find *any* discrepancies.

VI. CASE STUDIES OF BUGS

In this section, we describe selected semantic and crash-inducing bugs that NEZHA found during our experiments.

A. ClamAV File Format Validation Bugs

As described in Section II, discrepancies in the file format validation logic across programs can have dire security implications. Here we highlight two critical bugs, where ClamAV fails to parse specially crafted ELF and XZ files and thus does not scan them, despite the fact that the programs that commonly execute/extract these types of files process them correctly. These bugs allow an attacker to launch evasion attacks against ClamAV by injecting malware into specially crafted files.

*1) **ELF** - Mishandling of Malformed Header:* According to the ELF specification [1], the ELF header contains the `e_ident[EI_CLASS]` field, which specifies the type of machine (32- or 64-bit) the ELF file is compiled to run on. Values greater than 2 for this field are left undefined.

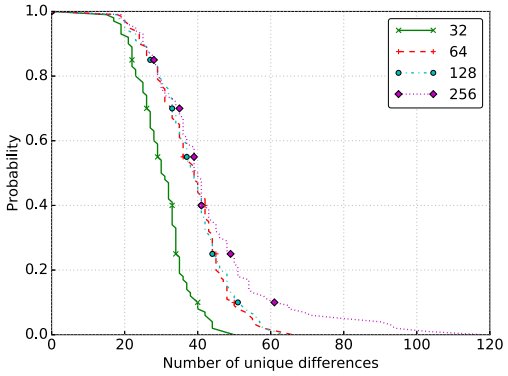

Fig. 10: Probability of finding at least n unique discrepancies across OpenSSL, LibreSSL, and BoringSSL with NEZHA running under output δ-diversity, for varying numbers of error codes, after $100,000$ executions (average of 100 runs, starting from a different seed corpus of 1000 certificates in each run).

In parsing ELF binaries, ClamAV differs from `binutils` when it encounters illegal values in `e_ident[EI_CLASS]`. As shown in Listing 1, ClamAV treats ELF binaries configured with such illegal values as being of an invalid format (`CL_EFORMAT`) and does not scan the respective files. By contrast, `binutils` correctly parses such ELF binaries. We verified that such ELF binaries can in fact be successfully executed. In Listing 2, the Linux kernel's ELF loader does not validate this field while loading a binary. As a result, a malware with such a corrupted ELF header can evade the detection of ClamAV, while retaining its capability to execute in the host OS.

```
1  static int cli_elf_fileheader(...) {
2    ...
3    switch(file_hdr->hdr64.e_ident[4]) {
4      case 1:
5        ...
6      case 2:
7        ...
8      default:
9        ...
10       return CL_EFORMAT;
```

Listing 1: ClamAV code that parses the `e_ident` field.

```
1  static int load_elf_binary(struct linux_binprm *bprm) {
2    ...
3    retval = -ENOEXEC;
4    if (memcmp(loc->elf_ex.e_ident, ELFMAG, SELFMAG) != 0)
5      goto out;
6    if (loc->elf_ex.e_type != ET_EXEC &&
         loc->elf_ex.e_type != ET_DYN)
7      goto out;
8    if (!elf_check_arch(&loc->elf_ex))
9      goto out;
10   ...
```

Listing 2: Error checks for ELF loading in the Linux kernel (the `e_ident` field is not checked).

*2) **XZ** - Mishandling of the Dictionary Size Field:* According to the XZ specifications [62], the LZMA2 decompression algorithm in an archive can use a dictionary size ranging from

4kB to 4GB. The dictionary size varies from file to file and is stored in the XZ header of a file. ClamAV differs from XZ Utils when parsing this dictionary size field.

```
1  extern lzma_ret lzma_lz_decoder_init(...) {
2    ...
3    // Allocate and initialize the dictionary.
4    if (next->coder->dict.size != lz_options.dict_size) {
5      lzma_free(next->coder->dict.buf, allocator);
6      next->coder->dict.buf
7        = lzma_alloc(lz_options.dict_size, allocator);
8      ...
9
10 lzma_alloc(size_t size, const lzma_allocator
       *allocator) {
11   ...
12   if (allocator != NULL && allocator->alloc != NULL)
13     ptr = allocator->alloc(allocator->opaque, 1, size);
14   else
15     ptr = malloc(size);
16   ...
```

Listing 3: XZ Utils parses the dictionary size correctly.

As shown in Listing 3, XZ Utils strictly conforms to the specifications and allocates a buffer based on the permitted dictionary sizes. On the other hand, ClamAV includes an additional check on the dictionary size that deviates from the specifications. It fails to parse archives with a dictionary size greater than 182MB (line 15 in Listing 4). As a result of this bug, when parsing such an archive containing a malware, ClamAV does not consider the file as an archive, and thus skips scanning the compressed malware.

```
1  SRes LzmaDec_Allocate(.., const Byte *props, ...) {
2    ...
3    dicBufSize = propNew.dicSize;
4    if (p->dic == 0 || dicBufSize != p->dicBufSize){
5      ...
6      // Invoke __xz_wrap_alloc()
7      p->dic = (Byte *)alloc->Alloc(alloc, dicBufSize);
8      if (p->dic == 0) {
9        ...
10       return SZ_ERROR_MEM;
11       ...
12
13 void *__xz_wrap_alloc(void *unused, size_t size) {
14   // Fails if size > (182*1024*1024)
15   if(!size || size > CLI_MAX_ALLOCATION)
16     return NULL;
17     ...
```

Listing 4: ClamAV's additional erroneous check on dictionary size.

B. X.509 Certificate Validation Discrepancies

In this Section, we present two examples of certificate validation semantic bugs found by NEZHA, one involving LibreSSL and one GnuTLS. Another example of a discrepancy between LibreSSL and BoringSSL is presented in the Appendix.

1) ***LibreSSL*** *- Incorrect parsing of time field types:* The RFC standards for X.509 certificates restrict the `Time` fields to only two forms, namely the ASN.1 representations of `UTCTime` (`YYMMDDHHMMSSZ`) and `GeneralizedTime` (`YYYYMMDDHHMMSSZ`) [15] which are 13 and 15 characters wide respectively. Time fields are also encoded with an ASN.1 tag that specifies their format. Despite the standards, in practice, we observe that 11- and 17-character time fields are used in the wild, by searching within the SSL observatory [7]. Indeed, some SSL libraries like OpenSSL and BoringSSL are more permissive while parsing such time fields.

LibreSSL, on the other hand, tries to comply strictly with the standards when parsing the validity time fields in a certificate. However, while doing so, LibreSSL introduces a bug. Unlike the other libraries, LibreSSL ignores the ASN.1 time format tag, and infers the time format type based on the length of the field (Lines 10 and 16 in Listing 5). In particular, the time fields in a certificate can be crafted to trick LibreSSL to erroneously parse the time fields using an incorrect type. For instance, when the time field of ASN.1 `GeneralizedTime` type is crafted to have the same length as the `UTCTime` (i.e., 13), LibreSSL treats the `GeneralizedTime` as `UTCTime`.

As a result of this confusion, LibreSSL may erroneously treat a valid certificate as not yet valid, when in fact it is valid; or, it may erroneously accept an expired certificate. For example, while other libraries may interpret a `GeneralizedTime` time in *history*, `201201010101Z` as `Jan 1 01:01:00 2012 GMT`, LibreSSL will incorrectly interpret this time as a `UTCTime` time in *future*, i.e., as `Dec 1 01:01:01 2020 GMT`. Note that finding time fields of non-standard lengths in the wild suggests that CAs do not actively enforce these standards length requirement. Furthermore, we also found certificates with `GeneralizedTime` times that are of the length 13 in the SSL observatory dataset.

```
1  int asn1_time_parse(..., size_t len, ..., int mode) {
2    ...
3    int type = 0;
4    /* Constrain to valid lengths. */
5    if (len != UTCTIME_LENGTH && len != GENTIME_LENGTH)
6      return (-1);
7    ...
8    switch (len) {
9    case GENTIME_LENGTH:
10     // mode is "ignored" -- configured to 0 here
11     if (mode == V_ASN1_UTCTIME)
12       return (-1);
13     ...
14     type = V_ASN1_GENERALIZEDTIME;
15   case UTCTIME_LENGTH:
16     if (type == 0) {
17       if (mode == V_ASN1_GENERALIZEDTIME)
18         return (-1);
19       type = V_ASN1_UTCTIME;
20     }
21     ...
```

Listing 5: `LibreSSL` time field parsing bug.

2) ***GnuTLS*** *- Incorrect validation of activation time:* As shown in Listing 6, GnuTLS lacks a check for cases where the year is set to 0. As a result, while other SSL libraries reject a malformed certificate causing t to be 0, GnuTLS erroneously accepts it.

```
1  static unsigned int check_time_status(gnutls_x509_crt_t
       crt, time_t now) {
2      int status = 0;
3      time_t t = gnutls_x509_crt_get_activation_time(crt);
4      if (t == (time_t) - 1 || now < t) {
5          status |= GNUTLS_CERT_NOT_ACTIVATED;
6          status |= GNUTLS_CERT_INVALID;
7          return status;
8      ...
```

Listing 6: GnuTLS activation time parsing error.

C. PDF Viewer Discrepancies

As mentioned in Section V-A, NEZHA uncovered 7 unique discrepancies in the tested three PDF browsers (Evince, Xpdf and MuPDF) over a total of 10 million generations. Examples of the found discrepancies include PDF files that could be opened in one viewer but not another and PDFs rendered with different contents across viewers. One interesting discrepancy includes a PDF that Evince treats as encrypted (thus opening it with a password prompt) but Xpdf recognizes as unencrypted (MuPDF and Xpdf abort with errors trying to render the file).

VII. RELATED WORK

Unguided Testing: Unguided testing tools generate test inputs independently across iterations without considering the test program's behavior on past inputs. *Domain-specific* evolutionary unguided testing tools have successfully uncovered numerous bugs across a diverse set of applications [2], [40], [42], [52], [56]. Another parallel line of work explored building different *grammar-based* testing tools that rely on a context free grammar for generating test inputs [48], [50]. LangFuzz [38] uses a grammar to randomly generate valid JavaScript code fragments and test JavaScript VMs, while GLADE [22] synthesizes a context-free grammar encoding the language of valid program inputs and leverages it for fuzzing. TestEra [49] uses specifications to automatically generate test inputs for Java programs. *lava* [58] is a domain-specific language designed for specifying grammars that can be used to generate test inputs for testing Java VMs. Unlike NEZHA's guided approach, the input generation process of these tools does not use any information from past inputs and essentially creates new inputs at random from a prohibitively large input space. This makes the testing process highly inefficient, since large numbers of inputs need to be generated to find a single bug.

Guided Testing: Evolutionary testing was designed to make the input generation process more efficient by taking program behavior information for past inputs into account, while generating new inputs [53]. Researchers have since explored different forms of code coverage heuristics (e.g., basic block, function, edge, or branch coverage) to efficiently guide the search for bug-inducing inputs. Coverage-based tools such as AFL [66], libFuzzer [4], and the CERT Basic Fuzzing Framework (BFF) [39] refine their input corpus by maximizing the code coverage with every new input added to the corpus. However, these tools are not well suited for differential testing as they do not exploit the relative differences across multiple test applications. In particular, to the best of our knowledge, NEZHA is the *first* testing framework to particularly design a path selection mechanism fitted towards to differential testing. Even if a state-of-the-art testing framework such as libFuzzer, was modified to perform differential testing using global coverage across multiple programs, it would still be outperformed by both NEZHA's gray-box and black-box engines, as shown in Section V.

Another line of research builds on the observation that the problem of new input generation from existing inputs can be modeled as a stochastic process. These tools leverage a diverse set of statistical techniques to drive input generation [23], [31], [47], or leverage static and dynamic analysis to prioritize deeper paths [55]. However, most of these tools do not support differential testing. Finally, Chen et al.'s tool perform differential testing of JVMs using MCMC sampling for input generation [31]. However, their tool is domain-specific (i.e., requires details knowledge of the Java class files and uses custom domain-specific mutations). Moreover, MCMC tends to be computationally very expensive, significantly slowing down the input generation process. NEZHA, by contrast, uses a fast guidance mechanism well suited for differential testing that seeks to maximize the diversity of *relative* behaviors of the test programs in search of discrepancies-inducing inputs.

Symbolic execution: Symbolic execution [43] is a white-box technique that executes a program *symbolically*, computes constraints along different paths, and uses a constraint solver to generate inputs that satisfy the collected constraints along each path. KLEE [26] uses symbolic execution to generate tests that achieve high coverage for several popular UNIX applications, however, due to path explosion, it does not scale to large applications. UC-KLEE [43], [54] aims to tackle KLEE's scalability issues by performing under-constrained symbolic execution, i.e., directly executing a function by skipping the whole invocation path up to that function. However, this may result in an increase in the number of false positives.

To mitigate path explosion, several lines of work utilize symbolic execution only in portions of their analysis to aid the testing process, and combine it with concrete inputs [27]. Another approach towards addressing the limitations of pure symbolic execution is to outsource part of the computation away from the symbolic execution engine using fuzzing [28], [34]–[37], [61]. A major limitation of symbolic-execution-assisted testing tools in the context of differential testing is that the path explosion problem increases significantly as the number of test programs increase. Therefore, it is very hard to scale symbolic execution techniques to perform differential testing of multiple large programs.

Differential Testing: Differential testing [51] has been very successful in uncovering semantic differences between independent implementations with similar intended functionality. Researchers have leveraged this approach to find bugs across many types of programs, such as web applications [29], different Java Virtual Machine (JVM) implementations [31], various security implementations of security policies for APIs [59], compilers [65] and multiple implementations of network protocols [25]. KLEE [26] used symbolic execution to perform differential testing, however suffers from scalability issues. SFADiff [21] performs black-box differential testing using Symbolic Finite Automata (SFA) learning, however, contrary to NEZHA, can only be applied to applications such as XSS filters that can be modeled by an SFA.

Chen et al. performed coverage-guided differential testing of SSL/TLS implementations using Mucerts [32]. However, unlike NEZHA, Mucerts requires knowledge of the partial grammar of the X.509 certificate format and applies MCMC algorithm on a *single* application (i.e., OpenSSL) to drive

its input generation. The input generation of Mucerts is very slow requiring multiple days to generate even 10,000 inputs. As demonstrated in Section V-A, NEZHA manages to find 27 times more discrepancies per input.

Another similar work is Brubaker et al.'s unguided differential testing system that synthesizes *frankencerts* by randomly combining parts of real certificates [24]. They use these syntactically valid certificates to test for semantic violations of SSL/TLS certificate validation across multiple implementations. However, unlike in NEZHA where the selection of mutated inputs is guided by δ-diversity metrics, the creation and selection of Frankencerts is completely unguided and therefore significantly inefficient compared to NEZHA.

Besides testing software, researchers have applied differential testing to uncover program deviations that could lead to malicious evasion attacks on security-sensitive programs. Similar to the way we applied NEZHA to uncover evasion bugs in ClamAV malware detector, Jana et al. use differential testing (with manually crafted inputs) to look for discrepancies in file processing across multiple antivirus scanners [41]. Recent works have applied differential testing to search for inputs that can evade machine learning classifiers for malware detection [46], [64]. However, unlike NEZHA, these projects require a detailed knowledge of the input format.

Differential testing shares parallels with N-version programming [30]. Both aim to improve the reliability of systems by using independent implementations of functionally equivalent programs, provided that the failures (or bugs) of the multiple versions are statistically independent. Therefore, NEZHA's input generation scheme will also be helpful to efficiently identify uncorrelated failures in software written under the N-version programming paradigm. Both N-version programming and differential testing suffer from similar limitations when different test programs demonstrate correlated buggy behaviors as observed by Knight et al. [44].

VIII. FUTURE WORK

We believe NEZHA is a crucial first step towards building efficient differential testing tools. However, several components of the underlying engine offer fertile ground for future work.

Mutation Strategies: NEZHA's current mutation strategies are not tailored for differential testing and therefore present a promising target for further optimization. Moreover, new gray-box guidance mechanisms that incorporate bookkeeping of intermediate states explored during a test program's execution could be used to more efficiently generate promising inputs.

Bug Localization: Similar improvements can be achieved towards the problem of automated debugging and bug localization. Prior research has performed bug bucketing for crash-inducing bugs using stack trace hashes [28]. However, this method is not suitable for semantic bugs that do not result in crashes. Moreover, heuristics such as using the average stack trace depth in order to locate "deeper" bugs cannot be trivially adapted to differential testing, because the depth of the root cause of a bug might not be correlated with the maximum depth of the execution. One possible solution for this problem is to utilize more complex schemes keeping track of all successful and failed executions across the tested applications (e.g., execution paths leading to successful and failed states may be stored in two distinct groups. Upon a deviation from a previously unseen behavior, one may lookup the point at which the deviation occurred in both groups to pinpoint the root cause.

IX. DEVELOPER RESPONSES

We have responsibly disclosed the vulnerabilities identified in this work to the respective developers of the affected programs. Each of our reports includes a description of the bug alongside a Proof-of-Concept (PoC) test input and a suggested patch. The wolfSSL team assigned the highest priority to all the memory corruption errors we reported and addressed all the bugs within six days of our disclosure, merging the respective patches in wolfSSL v3.9.8. Likewise, ClamAV developers have confirmed the reported bugs and are planning to merge the relevant fixes in v0.99.3. The ClamAV evasions bugs have been assigned with CVE identifiers CVE-2017-6592 (XZ archive evasion) and CVE-2017-6593 (ELF binary evasion). GnuTLS and LibreSSL developers likewise addressed the reported bugs within three days from our disclosure, pushing the respective patches to upstream.

X. CONCLUSION

In this paper we design, implement, and evaluate NEZHA, a guided differential testing tool that realizes the concept of δ-diversity to efficiently find semantic bugs in large, real-world applications without knowing any details about the input formats. NEZHA can generate test inputs using both δ-diversity black-box and gray-box guidance. Our experimental results demonstrate that NEZHA is more efficient at finding discrepancies than all of the guided and unguided testing frameworks we compared it against. NEZHA discovered two evasion attacks against the ClamAV malware detector and 764 discrepancies between the implementations of X.509 certificate validation in six major SSL/TLS libraries.

We have made NEZHA open-source so that the community can continue to build on it and advance the field of efficient differential testing for security bugs. The framework can be accessed at https://github.com/nezha-dt.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback. This work is sponsored in part by the Office of Naval Research (ONR) through contract N00014-15-1-2180 and by the National Science Foundation (NSF) grants CNS-13-18415 and CNS-16-17670. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, ONR or NSF.

REFERENCES

[1] "Executable and Linkable Format (ELF)," http://www.skyfree.org/linux/references/ELF_Format.pdf.

[2] "Ioactive_elf_parsing_with_melkor.pdf," http://www.ioactive.com/pdfs/IOActive_ELF_Parsing_with_Melkor.pdf.

[3] "Isartor test suite (terms of use & download) - pdf association," https://www.pdfa.org/isartor-test-suite-terms-of-use-download/.

[4] "libFuzzer - a library for coverage-guided fuzz testing - LLVM 3.9 documentation," http://llvm.org/docs/LibFuzzer.html.

[5] "Nezha (chinese protection god)," http://www.godchecker.com/pantheon/chinese-mythology.php?deity=NEZHA.

[6] "Santizercoverage - Clang 4.0 documentation," http://clang.llvm.org/docs/SanitizerCoverage.html.

[7] "The EFF SSL Observatory," https://www.eff.org/observatory.

[8] "Undefined behavior sanitizer - Clang 4.0 documentation," http://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[9] "Virusshare.com," https://virusshare.com/.

[10] "Internet X.509 public key infrastructure certificate policy and certification practices framework," http://www.ietf.org/rfc/rfc2527.txt, 1999.

[11] "The TLS protocol version 1.0," http://tools.ietf.org/html/rfc2246, 1999.

[12] "HTTP over TLS," http://www.ietf.org/rfc/rfc2818.txt, 2000.

[13] "System v application binary interface," https://refspecs.linuxfoundation.org/elf/gabi4+/contents.html, April 2001.

[14] "The Transport Layer Security (TLS) protocol version 1.1," http://tools.ietf.org/html/rfc4346, 2006.

[15] "Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile," http://tools.ietf.org/html/rfc5280, 2008.

[16] "The Transport Layer Security (TLS) protocol version 1.2," http://tools.ietf.org/html/rfc5246, 2008.

[17] "Representation and verification of domain-based application service identity within Internet public key infrastructure using X.509 (PKIX) certificates in the context of Transport Layer Security (TLS)," http://tools.ietf.org/html/rfc6125, 2011.

[18] "The Secure Sockets Layer (SSL) protocol version 3.0," http://tools.ietf.org/html/rfc6101, 2011.

[19] "Xz utils," http://tukaani.org/xz/, 2015.

[20] "The Transport Layer Security (TLS) Protocol Version 1.3," https://tools.ietf.org/html/draft-ietf-tls-tls13-14, 2016.

[21] G. Argyros, I. Stais, S. Jana, A. D. Keromytis, and A. Kiayias, "SFADiff: Automated evasion attacks and fingerprinting using black-box differential automata learning," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2016, pp. 1690–1701.

[22] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing program input grammars," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2017.

[23] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, 2016, pp. 1–12.

[24] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov, "Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2014, pp. 114–129.

[25] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song, "Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation," in *16th USENIX Security Symposium (USENIX Security '07)*. USENIX Association, 2007.

[26] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, vol. 8, 2008, pp. 209–224.

[27] C. Cadar and D. Engler, "Execution generated test cases: How to make systems code crash itself," in *International SPIN Workshop on Model Checking of Software*. Springer, 2005, pp. 2–23.

[28] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *2015 IEEE Symposium on Security and Privacy (S&P)*, May 2015, pp. 725–741.

[29] P. Chapman and D. Evans, "Automated black-box detection of side-channel vulnerabilities in web applications," in *Proceedings of the 18th ACM conference on Computer and Communications Security (CCS)*. ACM, 2011, pp. 263–274.

[30] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," in *Digest of Papers FTCS-8: Eighth Annual International Conference on Fault Tolerant Computing*, 1978, pp. 3–9.

[31] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of JVM implementations," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2016, pp. 85–99.

[32] Y. Chen and Z. Su, "Guided differential testing of certificate validation in SSL/TLS implementations," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (FSE)*. ACM, 2015, pp. 793–804.

[33] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: validating SSL certificates in non-browser software," in *Proceedings of the 2012 ACM conference on Computer and Communications Security (CCS)*. ACM, 2012, pp. 38–49.

[34] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008, pp. 206–215.

[35] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, vol. 40, no. 6. ACM, 2005, pp. 213–223.

[36] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *Proceedings of the 2008 Network and Distributed Systems Symposium (NDSS)*, vol. 8, 2008, pp. 151–166.

[37] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *22nd USENIX Security Symposium (USENIX Security '13)*. Washington, D.C.: USENIX, 2013, pp. 49–64.

[38] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *21st USENIX Security Symposium (USENIX Security '12)*, 2012, pp. 445–458.

[39] A. D. Householder and J. M. Foote, "Probability-based parameter selection for black-box fuzz testing," in *CMU/SEI Technical Report - CMU/SEI-2012-TN-019*, 2012.

[40] S. Jana, Y. Kang, S. Roth, and B. Ray, "Automatically Detecting Error Handling Bugs using Error Specifications," in *25th USENIX Security Symposium (USENIX Security)*, Austin, August 2016.

[41] S. Jana and V. Shmatikov, "Abusing file processing in malware detectors for fun and profit," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2012, pp. 80–94.

[42] Y. Kang, B. Ray, and S. Jana, "APEx: Automated Inference of Error Specifications for C APIs," in *31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Singapore, September 2016.

[43] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[44] J. C. Knight and N. G. Leveson, "An experimental evaluation of the assumption of independence in multiversion programming," *IEEE Transactions on Software Engineering*, no. 1, pp. 96–109, 1986.

[45] J. Kornblum, "Identifying almost identical files using context triggered piecewise hashing," *Digital Investigation*, vol. 3, pp. 91–97, 2006.

[46] P. Laskov *et al.*, "Practical evasion of a learning-based classifier: A case study," in *2014 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2014, pp. 197–211.

[47] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, vol. 50, no. 10. ACM, 2015, pp. 386–399.

[48] B. A. Malloy and J. F. Power, "An interpretation of purdom's algorithm for automatic generation of test cases," in *International Conference on Computer and Information Science*, 2001.

[49] D. Marinov and S. Khurshid, "Testera: A novel framework for automated testing of java programs," in *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE)*.

Washington, DC, USA: IEEE Computer Society, 2001, pp. 22–. [Online]. Available: http://dl.acm.org/citation.cfm?id=872023.872551
[50] P. M. Maurer, "Generating test data with enhanced context-free grammars," *IEEE Software*, vol. 7, no. 4, pp. 50–55, 1990.
[51] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.
[52] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.
[53] R. P. Pargas, M. J. Harrold, and R. R. Peck, "Test-data generation using genetic algorithms," *Software Testing Verification and Reliability*, vol. 9, no. 4, pp. 263–282, 1999.
[54] D. A. Ramos and D. R. Engler, "Practical, low-effort equivalence verification of real code," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 669–685.
[55] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.
[56] J. Ruderman, "Introducing jsfunfuzz," https://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/.
[57] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: a fast address sanity checker," in *2012 USENIX Annual Technical Conference (USENIX ATC 2012)*, 2012, pp. 309–318.
[58] E. G. Sirer and B. N. Bershad, "Using production grammars in software testing," in *Proceedings of the 2nd conference on Domain-Specific Languages (DSL)*, vol. 35, no. 1. ACM, 1999, pp. 1–13.
[59] V. Srivastava, M. D. Bond, K. S. McKinley, and V. Shmatikov, "A security policy oracle: Detecting security holes using multiple api implementations," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 343–354, 2011.
[60] E. Stepanov and K. Serebryany, "Memorysanitizer: fast detector of uninitialized memory use in C++," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, 2015, pp. 46–55.
[61] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.
[62] Tool Interface Standard, "The .xz File Format," http://tukaani.org/xz/xz-file-format.txt, August 2009.
[63] Tool Interface Standard (TIS), "Executable and Linking Format (ELF) specification," https://refspecs.linuxfoundation.org/elf/elf.pdf, May 1995.
[64] W. Xu, Y. Qi, and D. Evans, "Automatically evading classifiers a case study on PDF malware classifiers," in *Proceedings of the 2016 Network and Distributed Systems Symposium (NDSS)*, 2016.
[65] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2011, pp. 283–294.
[66] M. Zalewski, "american fuzzy lop," http://lcamtuf.coredump.cx/afl/.
[67] A. Zeller, "Yesterday, my program worked. Today, it does not. Why?" in *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. Springer, 1999, pp. 253–267.

XI. Appendix

A. Memory Coruption Bugs Reported by NEZHA

1) ClamAV use-after-free: NEZHA disclosed a use-after-free heap bug in ClamAV, which is invoked when parsing a malformed XZ archive. As ClamAV parses the multiple compression blocks in the archive, it makes a series of allocation and freeing operations on a single memory buffer. ClamAV's memory *allocation* routine will only do so when the given memory pointer is `NULL`. However, the memory *freeing* routine fails to nullify the memory pointer after freeing the buffer. As a result, the bug will be triggered after a series of allocate-free-allocate operations. An attacker can exploit this vulnerability by sending a malformed XZ archive that will crash ClamAV when ClamAV attempts to scan the archive.

2) wolfSSL memory errors: NEZHA uncovered four memory corruption bugs in wolfSSL, all of which were marked as critical by the wolfSSL developers and patched within six days after we reported the bugs. Two of the bugs were caused by missing checks for malformed PEM certificate headers inside the `PemToDer` function, which converts a X.509 certificate from PEM to DER format. The missing checks resulted in out-of-bounds memory reads. The third bug was caused by a missing check for the return value of a `PemToDer` call, inside the `wolfSSL_CertManagerVerifyBuffer` routine, causing a segmentation fault. In this case, the structure holding the DER-converted certificate is corrupted. Finally the fourth bug, also occurring inside `Pem2Der`, resulted in an out-of-bounds read, due to a missing check on the size of the PEM certificate to be converted. This can be triggered by an intermediate certificate in a chain that has the correct PEM header but an empty body: the missing check will cause `Pem2Der` to not return any error, which in turn results in an out-of-bounds memory access during the subsequent steps of the verification process.

3) GnuTLS null pointer dereference: NEZHA found a missing check inside the `gnutls_oid_to_ecc_curve` routine of GnuTLS, where dereferenced pointers were not checked to be not `NULL`. This bug resulted in a segmentation fault while parsing an appropriately crafted certificate.

B. Coverage and population size for NEZHA's different guidance engines

In Figures 11 and 12, we present the coverage and population increases for the different engines of NEZHA for the experimental setup of Section V-A.

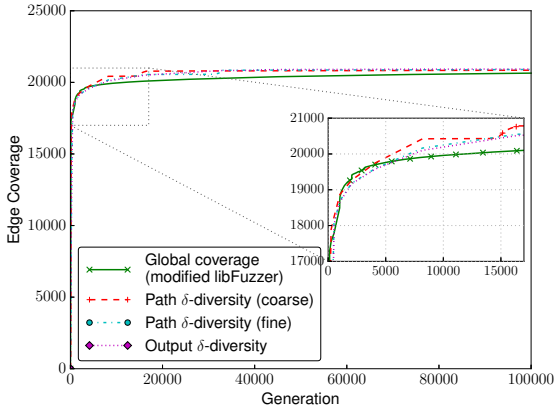

Fig. 11: Coverage increase for each of NEZHA's engines per generation (average of 100 runs with a seed corpus of 1000 certificates).

C. *BoringSSL* - Incorrect representation of `KeyUsage`

According to the RFC standards, the `KeyUsage` extension defines the purpose of the certificate key and it uses a `bitstring` to represent the various uses of the key. A valid

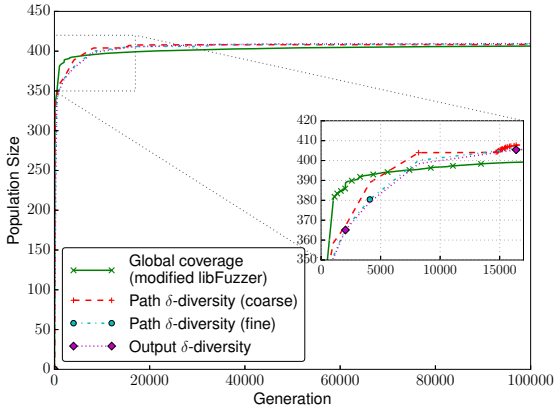

Fig. 12: Population size increase for each of NEZHA's engines per generation (average of 100 runs, each starting from a seed corpus of 1000 certificates).

Certificate Authority (CA) certificate must have this extension present with the `keyCertSign` bit set.

BoringSSL and LibreSSL differ in the way they parse the ASN.1 `bitstring`, which is used for storing the `KeyUsage` extension in the X.509 certificates. Each `bitstring` is encoded with a "padding" byte that indicates the number of least significant unused bits in the bit representation of the structure. This byte should never be more than 7. But if the byte is set to a value greater than 7, BoringSSL fails to parse the bitstring and throws an error in Listing 7, whereas LibreSSL masks that byte with `0x07` and continues to parse the bitstring as-is as shown in Listing 8.

```
ASN1_BIT_STRING *c2i_ASN1_BIT_STRING(..., char **pp) {
  ...
  p = *pp;
  padding = *(p++);
  // returns an error if invalid padding byte
  if (padding > 7) {
    OPENSSL_PUT_ERROR(ASN1,
        ASN1_R_INVALID_BIT_STRING_BITS_LEFT);
    goto err;
  }
  ret->flags &= ~(ASN1_STRING_FLAG_BITS_LEFT | 0x07);
  ret->flags |= (ASN1_STRING_FLAG_BITS_LEFT  | i);
  ...
```

Listing 7: BoringSSL code for validating `bitstrings`.

```
ASN1_BIT_STRING *c2i_ASN1_BIT_STRING(..., char **pp) {
  ...
  p = *pp;
  i = *(p++);
  // masks the padding byte, instead of with a check
  ret->flags&= ~(ASN1_STRING_FLAG_BITS_LEFT| 0x07);
  ret->flags|=(ASN1_STRING_FLAG_BITS_LEFT  | (i&0x07));
  ...
```

Listing 8: LibreSSL code for validating `bitstrings`.

This subtle discrepancy results in two different interpretations of the same bitstring used in the extension. BoringSSL fails to parse the bitstring and results in an empty `KeyUsage` extension. LibreSSL, by masking the padding byte, successfully parses the extension. We also find that these libraries exhibit this discrepancy during the parsing of a Certificate Signing Request (CSR). This can have critical security implications. Consider the scenario where a CA using BoringSSL parses such a CSR presented by an attacker and does not interpret the extension correctly. The CA misinterprets the key usages and does not detect certain blacklisted ones. In this situation, the CA might copy the malformed extension to the issued certificate. Subsequently, when the issued certificate is parsed by a client using LibreSSL, it will be parsed with a valid keyUsage extension and thus the attacker can use the certificate for purposes that were not intended by the CA.