Neuro-Symbolic Execution: Augmenting Symbolic Execution with Neural Constraints

Shen Shiqi Shweta Shinde Soundarya Ramesh Abhik Roychoudhury Prateek Saxena
Computer Science Department, School of Computing
National University of Singapore
{shiqi04, shweta24, sramesh, abhik, prateeks}@comp.nus.edu.sg

***Abstract*—Symbolic execution is a powerful technique for program analysis. However, it has many limitations in practical applicability: the path explosion problem encumbers scalability, the need for language-specific implementation, the inability to handle complex dependencies, and the limited expressiveness of theories supported by underlying satisfiability checkers. Often, relationships between variables of interest are not expressible directly as purely symbolic constraints. To this end, we present a new approach—*neuro-symbolic execution*—which learns an approximation of the relationship between program values of interest, as a neural network. We develop a procedure for checking satisfiability of *mixed* constraints, involving both symbolic expressions and neural representations. We implement our new approach in a tool called NEUEX as an extension of KLEE, a state-of-the-art dynamic symbolic execution engine. NEUEX finds 33 exploits in a benchmark of 7 programs within 12 hours. This is an improvement in the bug finding efficacy of 94% over vanilla KLEE. We show that this new approach drives execution down difficult paths on which KLEE and other DSE extensions get stuck, eliminating limitations of purely SMT-based techniques.**

I. INTRODUCTION

Symbolic execution is a code analysis technique which reasons about sets of input values that drive the program to a specified state [68]. Certain inputs are marked as symbolic and the analysis gathers symbolic constraints on these inputs, by analyzing the operations along a path of a program. Satisfying solutions to these constraints are concrete values that cause the program to execute the analyzed path. Manipulating these constraints allows one to reason about the reachability of different paths and states, thereby serving to guide search in the execution space efficiently. Symbolic execution, especially its mixed-dynamic variant, has been widely used in computer security. Its prime application over the last decade has been in white-box fuzzing, with the goal of discovering software vulnerabilities [63]. More broadly, it has been used for patching [80], invariant discovery [66], and verification to prove the absence of vulnerabilities [45]. Off-the-shelf symbolic execution tools targeting languages such as C/C++ [95], JavaScript [73], and executable binary code [44] are available.

Symbolic analysis is a powerful technique; however, it has a number of limitations in practical applicability. First, symbolic analysis is a deductive procedure, requiring complete access to target code and pre-specified semantics of the target language. The symbolic analysis procedure is specific to the target language (e.g., C vs. x64), and further, if a certain functionality of a program is unavailable for analysis—either because it is implemented in a different language, or because it is accessible as a closed, proprietary service—then, such functionality cannot be analyzed precisely. Today's symbolic execution engines either resort to human assistance (e.g., prompting analysts for external stubs) or to ad-hoc concretization of symbolic values in such cases.

Second, symbolic analyses may not be able to infer program constraints that are directly expressible in underlying SAT/SMT theories succinctly. Programs often have custom logic that implements high-level relations (e.g., string manipulation) via low-level control flow constructs (e.g., iterating over byte arrays in nested loops). This often contributes to the phenomenon described as the "path explosion" problem, wherein the symbolic analysis enumeratively explores the execution path space which can be exponentially large [41]. Providing structured representation of constraints (e.g., arithmetic expressions or strings) is one approach that has yielded improvements [33], [56], [91], [100]. However, recovering and solving such structured constraints require specialized techniques, each targeted at a specific class of constraints.

Lastly, many symbolic constraints, even if recovered succinctly, lack efficient handling or fall outside of the theory of the underlying SAT/SMT checkers [34]. Symbolic analysis typically uses quantifier-free and decidable theories in first-order logic, and satisfiability solvers have well-known limits in expressiveness [30]. For instance, non-linear arithmetic over reals is slow and does not scale in existing solvers [57], and string support is relatively new and still an area of active research [100]. When program functionality does not fall within the supported theories, analysis either precludes such functionality altogether, encodes it abstractly using supported theories (e.g., arrays, bit-vectors, or uninterpreted functions), or concretizes symbolic variable with ad-hoc values.

A. Neuro-Symbolic Execution

In this paper, we introduce a new approach that complements dynamic symbolic execution and is easily implementable in standard tools. We present a technique called *neuro-symbolic execution*, in which the symbolic execution engine accumulates *neural* constraints (learned inductively) for certain parts of the analyzed program logic, in addition to its standard *symbolic* constraints (derived deductively). Neural constraints capture relationships between program values for

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23530
www.ndss-symposium.org

which the symbolic engine cannot recover quickly solvable constraints. In neuro-symbolic execution, the engine can switch to an inductive learning mode by forking off a live training procedure for such program logic. The training procedure treats the target logic as a black-box and learns a neural network representation approximating it as accurately as feasible. This learnt neural network is called a neural constraint, and both symbolic and neural constraints are called *neuro-symbolic*.

Our choice of representation via neural networks is motivated by two observations. First, neural networks can approximate or represent a large category of functions, as implied by the universal approximation theorem [55]; and in practice, an explosion of empirical results are showing that they are learnable for many practical functions [32], [64]. Although specialized training algorithms are continuously on the rise [69], [85], we expect that neural networks will prove effective in learning approximations to several useful functions we encounter in practice. Second, neural networks are a differentiable representation, often trained using optimization methods such as gradient descent [87]. This differentiability allows for efficient analytical techniques to check for satisfiability of neural constraints, and produce satisfying assignments of values to variables [65], analogous to the role of SMT solvers for purely symbolic constraints. One of the core technical contributions of this work is a procedure to solve neuro-symbolic constraints: checking satisfiability and finding assignments for variables involved in neural and symbolic constraints simultaneously, with good empirical accuracy on tested benchmarks. This is the key to utilizing neuro-symbolic execution for bug-finding.

B. Applications, Tool & Results

In this work, we focus on establishing the benefits of augmenting dynamic symbolic execution with neural constraints for one security application: *generating exploits* for out-of-bound buffer accesses, zero division, and data-type overflows. In this setting, we take a well-maintained dynamic symbolic execution engine called KLEE as a baseline [40], and augment it to switch to neuro-symbolic execution for parts of the code if it gets stuck because of path explosion, solver timeouts, or external calls. We call this enhanced tool NeuEx.

Results. We analyze 7 real-world Linux programs for the three classes of vulnerabilities listed above. Our tool NeuEx, which extends KLEE, finds a total of 33 bugs in 12 hours. 11 of these are new and 22 have publicly known CVEs. Vanilla KLEE (without our neuro-symbolic enhancement) finds only 17 of these in 12 hours; so NeuEx improves over the baseline by finding 94% more bugs. We show that NeuEx helps vanilla KLEE as it drives program execution down complex paths on which the latter gets stuck. We show that NeuEx scales gracefully with the increasing complexity of constraints; it learns both simple constraints (where KLEE is fast) as well as complex ones (which KLEE times out in 12 hours). We compare NeuEx with a structured constraint inference extension of symbolic execution called LESE [91], finding that NeuEx is two orders of magnitudes faster.

Contributions. We make the following contributions:

- *Neuro-Symbolic Constraints.* NeuEx is the first inductive approach that uses to learn an approximate representation of difficult program path logic in symbolic execution. This is a generic approach to learn a representation of constraints different from those encoded in the program implementation or recovered by prior template-based inductive approaches.
- *Neuro-Symbolic Constraint Solving.* NeuEx features a novel procedure that solves purely symbolic, purely neural, and mixed neuro-symbolic constraints.
- *Tool and Evaluation.* Our approach can augment existing tools, which we confirm by extending a state-of-the-art dynamic symbolic execution system. Our evaluation confirms that our new approach directly alleviates known challenges for deductive symbolic analysis: path explosion, limitations of SMT theories, and missing external code.

```
1  #define BUFFER_LEN 4096
2  void psf_log_printf (...) {
3   ...
4   while (...) { // KLEE path explosion
5   ...
6   }
7  }
8  double psf_calc_signal_max (SNDFILE *psf) {
9   ...
10  sf_read_double ((SNDFILE*) psf, data, .. );
11  ...
12  temp = fabs (data [k]); // KLEE cannot reason about fabs
13  ...
14  return temp;
15 }
16 void sfe_copy_data_fp(..., SNDFILE *infile,..) {
17  static double data[BUFFER_LEN], max;
18  ...
19  max = psf_calc_signal_max (infile);
20  while (readcount > 0) {
21    readcount = sf_readf_double(infile,data,frames);
22    for (k = 0; k < readcount; k++)
23      data[k] /= max; // potential divide-by-zero
24    ...
25  }
26 ...
27 }
28 int main (int argc, const char* argv[]) {
29  char* infilename = argv [argc-2];
30  ...
31  if (strlen (infilename) > 1 && infilename [0] == '-') {
32    // exit
33  }
34  ...
35  psf_log_printf(...);
36  ...
37  sfe_copy_data_fp(...);
38  ...
39 }
```

Fig. 1. A simplified example of libsndfile library. Function `sfe_copy_data_fp` copies data from input to output files. The function first scans the entire input file to obtain the maximum value of the signal (Line 19) by reading the input file bytes frame by frame in form of double values (Line 12). It then normalizes the values (Line 23), and writes the new frame values to the output file. The code has a potential divide by zero on Line 23.

II. Problem

We use KLEE, an existing and widely-used engine, as a baseline to show examples where symbolic execution exhibits limitations. Figure 1 shows a code snippet from an audio processing library called libsndfile [52], which has a divide-by-zero (Line 23) reported as CVE-2017-14246 [21].

Constraint Inference is Difficult. For KLEE to find the divide-by-zero, it has to infer two sets of *path constraints*.

First, the execution must reach Line 23 in Figure 1 by satisfying *reachability constraints*. Second, the value of the variable in the denominator of the division operation must be set to 0, which satisfies certain *vulnerability constraints* that result in a crash. Collecting reachability constraints for Line 23 involves getting past the complex grammar of the file header and selecting the right command-line options. Even after part of the reachability challenge is side-stepped by providing an input grammar [61], symbolic execution may not be able to collect the exact reachability and vulnerability constraints. Thus, the main challenge that still remains is to infer and solve all the constraints leading to value 0 for the denominator.

KLEE's classical DSE mode begins from the `main` function in the program (Line 28). We expect it to identify the vulnerability on Line 23 in function `sfe_copy_data_fp`. We mark the input file name and content as symbolic. The DSE mode gets past the branch which performs checks on the variable `infilename` (Line 31). However, the DSE mode gets stuck in `psf_log_printf` called on Line 35. This function has a complicated loop. The loop guards are controlled by symbolic values read from the file (Line 4). Even if KLEE gets past the loop in `psf_log_printf`, there are more than 10 loops (not shown in the Figure) in the call graph before reaching the `sfe_copy_data_fp` function. In our experiments, KLEE could not reach the vulnerability within a 12-hour timeout.

When KLEE encounters the complex loops in our example, it attempts to enumerate the entire path space of the low-level implementation, running into memory exhaustion. A human analyst can inspect and recognize that the constraint is perhaps representable differently—for instance, a part of the complexity can be eliminated from direct use of floating point values (doubles) for inputs which can then be subject to SMT reasoning. Generic approaches to tackling the "path explosion" phenomenon would not help recover such a representation automatically in this example. For example, a technique called loop-extended symbolic execution (LESE) [91] attempts to extract linear relationships between a small number of loop induction variables. However, in our example, the relationship is multi-linear and has floating point arithmetic, which does not fit the template expected by LESE. More powerful techniques like Veritesting [33] generalize to capture dependencies in multi-path code fragments as much richer SMT constraints. In the case of Veritesting, loops are unrolled to fixed depths and limited to certain function boundaries. However, one can see that SMT constraint encoding simply offload the exponential complexity to SMT solver. Therefore, while being useful over purely dynamic symbolic execution, prior approaches do not fundamentally lift the abstraction at which variable relationships are reasoned about. These approaches simply model relationships inherent in the low-level implementation as certain templatized constraint formats.

The key takeaway from this example is that learning the right representation of the constraint is a challenge. In prior constraint synthesis works, the representation of the constraints is fixed in advance as *templates*, such as in linear arithmetic [91], octagonal inequalities [82], or even full SMT theories [33]. Each templating technique comes with specialized inference procedure. When the code being approximated does not fall within chosen template structure, brute-force enumeration of templates to fit the samples or ad-hoc concretization is the default option undertaken in most prior works. This motivates our new approach to learn a different (approximate) representation of program fragments.

```
1  static int _tiffMapProc
2  (thandle_t fd, void** pbase, toff_t* psize){
3    ...
4    *pbase = mmap(0, (size_t)sizem, PROT_READ, MAP_SHARED,
          fdh.fd, 0); // returns -1
5    ...
6  }
7  // KLEE Stub
8  void *mmap(void *start, size_t length, int prot, int flags
       , int fd, off_t offset) {
9    klee_warning("ignoring (EPERM)");
10   errno = EPERM;
11   return (void*) -1;
12 }
```

Fig. 2. A simplified example of `libtiff` library. KLEE's uClibc stub does not model the behavior of `mmap` and merely returns an error (Line 11).

```
1  static enum req_action req_iquery(HEADER *hp, u_char **cpp
       , u_char *eom, int *buflenp, u_char *msg) {
2    if ((n = dn_skipname(*cpp, eom)) < 0){ // unknown call
3      return (Finish); // klee exit
4    }
5    ...
6    memcpy(anbuf, fname, alen); // buffer overflow
7    ...
8  }
9  // KLEE Stub
10 void __stub1(void) {
11   return;
12 }
13 link_warning (__stub1, "the `libresolv' library is a stub.
       Do you really need it?")
```

Fig. 3. A simplified example of BIND utility. `req_iquery` (Line 1) parses the DNS iquery packet to decides what response to send for the query. The program has a buffer overflow on Line 6. The logic before this line calls a `libc` function `dn_skipname` which is implemented in `libresolv` library external to the program. KLEE's uClibc library does not reason about any functions in this external library (Line 11), and always returns an error value. KLEE terminates its analysis on Line 3 and never reaches Line 6.

Constraint Solving is Difficult. Consider the scenario where KLEE or its extensions are somehow precisely able to infer the reachability and vulnerability constraints in the above example, encoded as SMT constraints over floating point values. At this point, the symbolic execution engine queries the SMT solver to check the satisfiability of these constraints to get concrete values for the symbolic variables. First, KLEE does not reason about floating point symbolic values and concretizes them to 0, thus leading to unsound SMT queries. Second, state-of-the-art SMT solvers which support floating point theories [27], [53] are well-known to be extremely slow [57]. One common option for solvers to handle difficult SMT theories, like floating points or strings, is to resort to bit-vector encoding and bit-blasting. In our example, this does not help. For instance, a bit-vector encoding of the floating point operations did not terminate in 12 hours with the Z3 SMT solver, which is re-confirmation of a known inscalability challenge [89].

Missing / Unreachable Code. Another fundamental limitation of deductive approaches to symbolic execution is that it requires access to all the source code under analysis. This poses a challenge in capturing complex dependency between variables, especially when the functions are implemented as external

libraries, remote calls, or a library call written in a different language. In Figure 1, there is an unknown external function call (`fabs`) whose output controls our variable of interest (Line 12). KLEE ships with a helper library uClibc [26] which provides stubs for reasoning about the most commonly used `libc` functions. The default uClibc does not define stubs for `fabs`. To deal with such missing code, the present practice is that developers have to manually analyze the program context and write stubs. We find many such instances of missing external calls. When analyzing libTIFF (Figure 2) with KLEE, it encounters the `mmap` library call which is not modeled by KLEE. Thus, it fails to analyze the interesting paths which have multiple CVEs [17], [23]–[25]. Figure 3 shows a code snippet from BIND [2] application, in which KLEE fails to find a known CVE because it cannot analyze `dn_skipname`. These examples motivate our technique which can approximate missing code with automatically learnt stubs, where possible.

III. Our Approach

To address the above challenges, we propose a new approach with two main insights: (1) leveraging the high representation capability of neural networks to learn constraints when symbolic execution is infeasible to capture it; and (2) encoding the symbolic constraints into neural constraint and leveraging the optimization algorithms to solve the neuro-symbolic constraints as a search problem. NeuEx departs from the purist view that all variable dependencies and relations should be *expressible precisely* in a symbolic form. Instead, NeuEx treats the entire code from Line 4-23 in Figure 1 as a black-box, and inductively learn a neural network —an approximate representation of the logic. In our example, we want the neural net to learn the relationship that the variable `max` is computed as the maximum quantity of all two-byte sequences in the input after being read in as positive double precision. Specifically, for each `i`, this desired relationship to be captured is:[1]

$$\begin{aligned} & \texttt{max} == ((2 \cdot \texttt{s} - 1) \cdot \texttt{a}) - 65536 \cdot (\texttt{s} - 1) \\ \wedge \quad & \texttt{a} == \texttt{infile}\,[\texttt{i}] + 256 \cdot \texttt{infile}\,[\texttt{i} + 1] \\ \wedge \quad & \texttt{s} == \texttt{sign}\,(\texttt{infile}\,[\texttt{i} + 1] \leq 127) \end{aligned}$$

This constraint when represented purely by a neural network approximation of the code is termed as a *neural constraint*. Our approach creates *neuro-symbolic* constraints, which includes both symbolic and neural constraints. The neural network is trained on concrete program values rather than on the code. Revisiting the example (Figure 1), the neuro-symbolic constraints capturing the vulnerability on Line 23 are:

$$\begin{aligned} & \texttt{strlen}(\texttt{infilename}) \leq 1 && (1) \\ \vee \quad & \texttt{infilename}\,[0] \neq 45 && (2) \\ \wedge \quad & \texttt{max} == 0.0 && (3) \\ \wedge \quad & \texttt{N} : \texttt{infile} \mapsto \texttt{max} && (4) \end{aligned}$$

(1)-(2) are symbolic constraints for reachability condition, while (3) is a symbolic constraint for the vulnerability condition divide-by-zero. (4) is a neural constraint capturing the relationship between the input and the variable of interest `max` in the divide-by-zero operation.

[1] $\texttt{infile}\,[\texttt{i}] + 256 \cdot \texttt{infile}\,[\texttt{i}+1]$ captures the relationship defined by function `sf_read_double` and $\texttt{max} == ((2 \cdot \texttt{s} - 1) \cdot \texttt{a}) - 65536 \cdot (\texttt{s} - 1)$ captures the behavior of the `fabs` function.

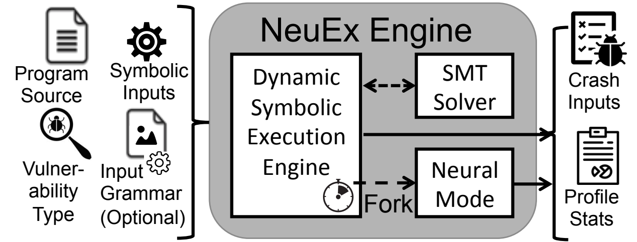

Fig. 4. NeuEx Component Architecture. tool takes in the program and optionally the symbolic inputs and the input grammar. The user can configure the types of vulnerabilities that tool should identify. tool runs in classic DSE mode and neural mode to produce concrete exploits.

Solving neuro-symbolic constraints is a key challenge. One naive way is to solve the neural and symbolic constraints separately. For example, consider the neuro-symbolic constraints in Equation (1)-(4). We can first solve the symbolic constraints by SAT/SMT solvers to obtain concrete values for variables `max`, `infilename`, and `infile`. Note that the SAT/SMT solver will assign a random value to `infile` since it is a free variable. When we plug the concrete value of `infile` from SAT/SMT solver in the neural constraint, it may produce values such as 32.00, 45.00, and 66.00 for the variable `max`. Although all these values of `max` satisfy the neural constraint, they may not satisfy the symbolic constraint `max == 0`. This discrepancy arises because we solve the symbolic and neural constraints individually without considering the inter-dependency of variables between them. We refer to such constraints with inter-dependent variables as *mixed constraints*. Alternatively, to solve these mixed constraints, one could resort to enumeration over values of the inter-dependent variables. However, this will require a lot of time to discovering the exploit. This inspires our design of neuro-symbolic constraint solving. NeuEx solves purely symbolic, purely neural, and mixed constraints in that order. Specifically, to solve the mixed constraints, NeuEx converts symbolic constraints to a loss function (or objective function) and then finds a satisfiable solution for neural constraints. This enables conjunction of symbolic and neural constraints.

Remark on Novelty. To the best of our knowledge, our approach is the first to train a neural net as a constraint and solve both symbolic constraint and neural constraint together. Inductive synthesis of symbolic constraints usable in symbolic analyses has been attempted in prior work [54], [81]. However, none of them use neural networks. Specifically, one notable difference is that our neural constraints are a form of *unstructured learning*, i.e., they approximate a large class of functions and do *not* aim to print out constraints in a symbolic form amenable to SMT reasoning. The main technical novelty of our approach is that the representation it learns is fundamentally different (approximation) from that of the real implementation. The second technical novelty in our design is that NeuEx reasons about neural as well as symbolic constraints simultaneously.

IV. Design

We first explain the NeuEx setup and the building blocks we use in our approach. Then, we present the core constraint solver of NeuEx along with various optimization strategies.

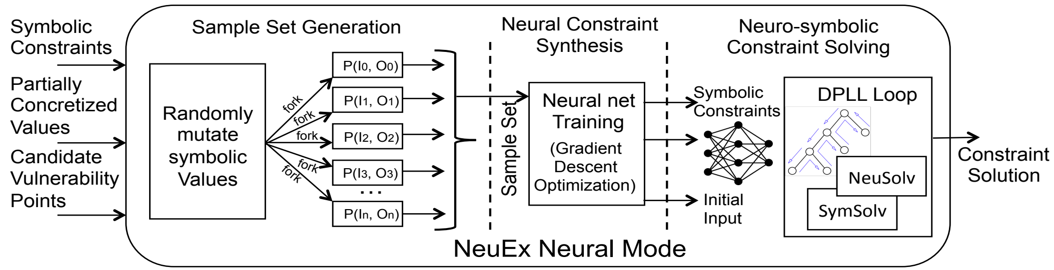

Fig. 5. NeuEx's Neural Mode. It takes in the intermediate symbolic constraints from the DSE mode. For sample set generation, it creates inputs and executes the program P in a separate forked process to produces tuples (I_n, O_n). It creates the train and test set from these tuples to learn a neural constraint for each CVP. The NeuEx solver then takes both neural as well as symbolic constraints and generates a concrete input if possible, else returns `UNSAT`.

A. Overview

Setup. NeuEx is built as an extension to KLEE—a widely used and maintained dynamic symbolic execution (DSE) engine. Figure 4 shows the architecture of NeuEx. It takes in the source code of the program that the user wants to analyze. The analyst can optionally mark inputs of interest as symbolic by standard KLEE interfaces. The analyst can further provide input grammar, which they know beforehand [35]. We are interested in identifying the following kinds of vulnerability conditions: out-of-bound buffer accesses, division by zero, and data-type overflows. To this end, KLEE symbolically executes the program. At the end of the analysis, KLEE returns concrete inputs to trigger all the detected vulnerabilities.

Preprocessing. NeuEx performs additional pre-processing of the program before starting the vulnerability detection. Specifically, it performs static analysis of the program source code to generate the call graph. NeuEx then statically estimates the program locations where (a) division operations may not check if the divisor is zero, and (b) buffer accesses may be without buffer boundary checks. Our analysis marks all such locations as *candidate vulnerability points (or CVPs)*. Lines 12, 23, and 29 will be marked as CVPs in Figure 1. For each CVP, NeuEx then statically instruments the programs to record the values of variables used in the statements at CVPs. NeuEx records the divisor for division operations; for buffer accesses it records the index used for dereference (e.g., `k`, `max`, and `argc` in Figure 1).

Classic DSE Mode. By default, NeuEx launches KLEE's standard DSE procedure which automatically constructs inputs required to drive the program execution to various CVPs. The DSE procedure begins at the entry point of the program. At each conditional branch, it logs the symbolic path constraints to reach this code point. It then invokes an SMT solver to obtain the concrete values for each symbolic variable in the constraint formula. The SMT solver returns concrete values for inputs if the given constraints are satisfiable; otherwise, it returns `UNSAT` which implies that the path is infeasible. The DSE procedure continues the path exploration to other branches. Once it reaches a CVP, it reports the concrete input values for an exploit which will trigger the bug. These can be concretely verified by running the program.

Neural Mode. We enhance KLEE with a profiler which monitors the classic DSE mode at runtime. It tracks 4 kinds of events, signaling that the DSE mode is stuck: (a) the path exploration is stuck in the same loop because of unrolling; (b) DSE runs out of memory due to path explosion; (c) the SMT solver is not able to find a `SAT`/`UNSAT` solution; and (d) DSE encounters an external / unknown function call. Whenever one of these events trigger, we terminate the DSE search on this path. Starting from the latest symbolic state, NeuEx forks separate parallel processes running each in a *neural mode*. This mode has a copy of all the symbolic constraints and concrete values up to this point in the execution.

Figure 5 shows the detailed steps of our neural mode. When the neural mode is triggered, NeuEx queries the call graph to identify all the CVPs which are statically reachable from the latest symbolic state. It chooses the nearest k CVPs of each type of bug, where k is a configurable parameter set to 150 by default. NeuEx treats the symbolic branch and each CVP location as a start and end point respectively, for training a neural network. NeuEx treats the fragment of code between the start and end points as a black-box which the neural network approximates.

To train the neural network, NeuEx needs to generate training samples. The samples consist of program values of variables at program entry and end point of the code fragment being approximated. NeuEx uses the symbolic constraints at the start point to generate concrete program inputs that lead up to that point using an SMT solver. This concrete input serves as a seed for generating many random mutations which are concretely executed to collect program values for training. Some of these mutated inputs reach the start and the end point. If sufficiently many samples (typically $100,000$) are collected, NeuEx uses it to train a neural network. In our example, the neural network collects samples for the values of the input file bytes at the start point of the program and the `max` variable at the end point (Line 23). At the end of such training, we have the symbolic constraints generated by DSE as well as a neural constraint, i.e., the neural net itself. Then NeuEx calls its solver to solve both these constraints simultaneously.

Constraint Solver. The NeuEx solver checks satisfiability of the given neuro-symbolic constraints and generates concrete values for constraint inputs. The syntax of our neuro-symbolic constraints is shown as an intermediate language in Table I. It is expressive enough to model various constraints specified in many real applications such as string and arithmetic constraints. Given the learned neuro-symbolic constraints, we seek the values of variables of interest that satisfy all the constraints within it. A key technical challenge is solving the mixed constraints simultaneously.

TABLE I. THE GRAMMAR OF NEURO-SYMBOLIC CONSTRAINT LANGUAGE SUPPORTED BY NEUEX.

Nuero-Symbolic Constraint	NS	:=	$N \wedge S$
Neural constraint	N	:=	$V_{I_n} \mapsto V_{O_n}$
symbolic constraint	S	:=	e1 $\ominus$ e2 | e
Variable	StrVar	:=	ConstStr | StrVar$\circ$StrVar
	NumVar	:=	ConstNum | NumVar$\oslash$NumVar
Expression	e	:=	contains(StrVar, StrVar) strstr(StrVar, StrVar) $\otimes$ NumVar strlen(StrVar) $\otimes$ NumVar NumVar $\otimes$ NumVar
Logical	$\ominus$	:=	$\vee$ | $\wedge$
Conditional	$\otimes$	:=	== | $\neq$ | > | $\geq$ | < | $\leq$
Arithmetic	$\oslash$	:=	+ | - | * | /

B. Constraint Learning

The procedure for learning neural networks is standard. Given a program, the selection of network architecture is the key for learning any neural constraint. In this paper, we use multilayer perceptron (MLP) architecture which consists of multiple layers of nodes and connects each node with all nodes in the previous layer [88]. Each node in the same layer does not share any connections with others. We select this architecture because it is a suitable choice for fixed-length inputs. There are other more efficient architectures (e.g., CNN [71] and RNN [77]) for the data with special relationships, and NEUEX is easily extensible to more network architectures.

The selection of activation function plays significant role for neural constraint inference as well. In this paper, we consider multiple activation functions (e.g., `Sigmoid` and `Tanh`) and finally select the rectifier function `Relu` as the activation function, because `Relu` obtains parse representation and reduces the likelihood of vanishing gradient [59]. In other words, the neural network with `Relu` has higher chance to converge than other activation functions.

In addition, to ensure the generality of neural constraint, we implement an early-stopping mechanism which is a regularization approach to reduce over-fitting [98]. It stops the learning procedure when the current learned neural constraint behaves worse on unseen test executions than the previous constraint. As the unseen test executions are never used for learning the neural constraint, the performance of learned neural constraint on unseen test executions is a fair measure for the generality of learned neural constraints.

NEUEX can use any machine learning approach, optimization algorithm (e.g., momentum gradient descent [85]) and regularization solution (e.g., dropout [96]) to learn the neural constraints. With future advances in machine learning, NEUEX can be adopted to new architectures and learning approaches.

C. Building Blocks for Solver

NEUEX's solves the neuro-symbolic constraints (e.g., Equations (1)-(4)) using its custom constraint solver detailed in Section IV-D. It uses two existing techniques as building blocks for such pure constraints: SMT solver and gradient-based neural solver. These solvers referred to as *SymSolv* and *NeuSolv* respectively form the basic building blocks.

SymSolv. NEUEX's symbolic constraint solver takes in first-order quantifier-free formulas over multiple theories (e.g., empty theory, the theory of linear arithmetic and strings) and returns `UNSAT` or concrete values as output. It internally employs Z3 Theorem Prover [53] as an SMT solver to solve both arithmetic and string symbolic constraints.

NeuSolv. For solving purely neural constraints, NeuSolv takes in the neural net and the associated loss function to generate the expected values that the output variables should have. NEUEX considers the neural constraint solving as a search problem and uses a gradient-based search algorithm to search for the satisfiable results [87]. Gradient-based search algorithm searches for the minimum of a given loss function $L(X)$ where X is a n-dimensional vector. The loss function can be any differentiable function that monitors the error between the objective and current predictions. Consider the example in Figure 1. By minimizing the error, NEUEX can discover the input closest to the exploit. To minimize the error, gradient-based search algorithm first starts with a random input X_0 which is the initial state of NeuSolv. For every enumeration i, it computes the derivative $\nabla_{X_i} L(X_i)$ given the input X_i and then updates X_i according to $\nabla_{X_i} L(X_i)$. This is based on the observation that the derivative of a function always points to a local nearest valley. The updated input X_{i+1} is defined as:

$$X_{i+1} = X_i - \epsilon \nabla_{X_i} L(X_i) \tag{5}$$

where ϵ is the learning rate that controls how much is going to be updated. Gradient-based search algorithm keeps updating the input until it reaches the local minima. To avoid the non-termination case, we set the maximum *number of enumerations* to be a constant M_e. If it exceeds M_e, NeuSolv stops and returns the current updated result. Note that the gradient-based search algorithm can only find the local minima since it stops when the error increases. If the loss function is a non-convex function with multiple local minima, the found local minima may not be the global minima. Moreover, it may find different local minima with different initial states. Thus, NEUEX executes the search algorithm multiple times with different randomized initial states in order to find the global minima of $L(X)$.

D. Constraint Solver

We propose a constraint solver for neuro-symbolic constraints with the help of SymSolv and NeuSolv. If the constraint solving procedure returns `SAT`, then the neuro-symbolic constraints are guaranteed to be satisfiable. It is not guaranteed, however, that the procedure terminates on all possible constraints; so, we bound its running time with a configurable timeout. Algorithm 1 shows the steps of our constraint solver.

DAG Generation. NEUEX takes the neuro-symbolic constraints and generates the directed acyclic graph (DAG) between constraints and its variables. Each vertex of the DAG represents a variable or constraint, and the edge shows that the variable is involved in the constraint. For example, Figure 6 shows the generated DAG for constraints $(V_1\ op_1\ V_2) \wedge (V_3\ op_2\ V_4) \wedge (V_5\ op_3\ V_6) \wedge (V_6\ op_3\ V_7\ op_4\ V_8) \wedge (V_8\ op_5\ V_9)$ where op_k can be any operator.

Next, NEUEX partitions the DAG into connected components by breadth-first search [39]. Consider the example

Algorithm 1 Algorithm for neuro-symbolic constraint solving. S_p is purely symbolic constraints; N_p is purely neural constraints; S_m and N_m are symbolic constraints and neural constraints in mixed components.

1: **function** NeuCL(S, N, MAX$_1$, MAX$_2$) ▷ S: Symbolic constraint list; N: Neural constraint list; MAX$_1$: The maximum number of trials of NeuSolv; MAX$_2$: The maximum number of trials for backtracking procedure.
2: (S_p, N_p, S_m, N_m) ← CheckDependency(N,S);
3: (X, assign1) ← SymSolv(S_p, $\emptyset$);
4: **if** X == UNSAT **then**
5: **return** (False, $\emptyset$);
6: **end if**
7: (X, assign2) ← NeuSolv(N_p);
8: assign ← Union(assign1, assign2);
9: ConflictDB ← $\emptyset$; trial_cnt ← 0;
10: **while** trial_cnt < MAX$_2$ **do**
11: ConflictConsts ← CreateConflictConsts(ConflictDB)
12: (X, assign3) ← SymSolv(S_m, ConflictConsts);
13: **if** X == UNSAT **then**
14: **go to** UNSAT
15: **end if**
16: NeuralConsts ← PartialAssign(N_m, assign3); cnt ← 0;
17: **while** cnt<MAX$_1$ **do**
18: (X, assign4) ← NeuSolv(NeuralConsts);
19: **if** X == SAT **then**
20: assign2 ← Union(assign3, assign4);
21: **go to** SAT
22: **end if**
23: cnt ← cnt+1;
24: **end while**
25: trial_cnt ← trial_cnt+1;
26: ConflictDB ← ConflictDB ∪ assign3;
27: **end while**
28: trial_cnt ← 0; F ← Encode(S_m, N_m);
29: **while** trial_cnt<MAX$_1$ **do**
30: assign2 ← NeuSolv(F);
31: X ← CheckSAT(assign2, S_m);
32: **if** X == SAT **then**
33: **go to** SAT
34: **end if**
35: trial_cnt ← trial_cnt+1;
36: **end while**
37: `UNSAT:`
38: **return** (False, $\emptyset$);
39: `SAT:`
40: **return** (True, Union(assign, assign2))
41: **end function**

shown in Figure 6. There are 5 constraints that are partitioned into three connected components, G_1, G_2, and G_3. NeuEx topologically sorts the components based on the type of constraints to schedule the solving sequence. Specifically, it clusters the components with only one kind of constraints as pure components (e.g., G_1 and G_2) and the components including both constraints as mixed components (e.g., G_3). It further sub-categorizes pure components into purely symbolic (e.g., G_1) and purely neural constraints (e.g., G_2).

NeuEx gives precedence to solving pure constraints over mixed constraints. This is because the constraints have different representation and hence are time-consuming to solve. Thus, in our example, NeuEx first solves $S_1 \wedge N_1$ and then checks the satisfiability of $S_2 \wedge N_2 \wedge S_3$.

Pure Constraint Solving. In pure constraints, we first apply SymSolv to solve purely symbolic constraints (Line 3) and then handle purely neural constraints using NeuSolv (Line 7). Note that the order of these two kinds of constraints does not affect the result. In our algorithm, we solve pure symbolic constraints first. If the SymSolv reports `UNSAT` for purely symbolic constraints, the whole neuro-symbolic constraints are `UNSAT`, as all the constraints are conjunctive.

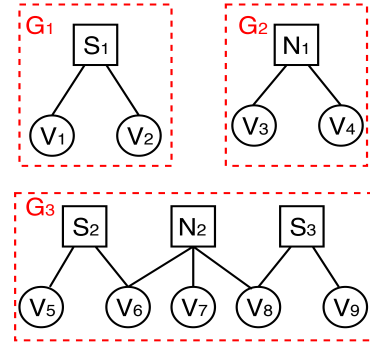

Fig. 6. NeuEx's DAG representation for neuro-symbolic constraints. The dotted rectangle represents connected components of the DAG. S: symbolic constraint; N: neural constraint; V: variable.

If both SymSolv and NeuSolv output `SAT`, NeuEx continues the process of solving the mixed constraints. We employ two strategies for solving mixed constraints based on the complexity of the constraint formulas. The first strategy is a greedy approach which works for simple relationships, but may fail after several attempts for complex relationships. If NeuEx cannot solve the formula with this strategy within a timeout, it switches to the second strategy.

Mixed Constraint Solving I. NeuEx obtains symbolic constraints from mixed components (e.g., S_2 and S_3) by cutting the edges between the neural constraints and its variables. Then, NeuEx invokes SymSolv to check their satisfiability on Line 12. If the solver returns `UNSAT`, NeuEx goes to `UNSAT` state; otherwise, NeuEx collects the concrete values of variables used in these symbolic constraints. Then, NeuEx plugs these concrete values into neural constraints on Line 16. For Figure 6 example, if the satisfiable result of $S_2 \wedge S_3$ is $\langle t_5, t_6, t_8, t_9 \rangle$ for the variables $\langle V_5, V_6, V_8, V_9 \rangle$, NeuEx partially assigns V_6 and V_8 in N_2 to be t_6 and t_8. Now, we have partially assigned neural constraint N_2' from N_2. All that remains is to search for the value of V_7 satisfying N_2'.

To solve such a partially assigned neural constraint, NeuEx employs NeuSolv on Line 18. If the NeuSolv outputs `SAT`, NeuEx goes to `SAT` state. In `SAT` state, NeuEx terminates and returns `SAT` with the combination of the satisfiable results for all the constraints. If the NeuSolv outputs `UNSAT`, NeuEx considers the satisfiable result of symbolic constraints as a counterexample and derives the conflict clauses on Line 11. Specifically, in our example NeuEx creates a new conflict clause $(V_5 \neq t_5) \vee (V_6 \neq t_6) \vee (V_8 \neq t_8) \vee (V_9 \neq t_9)$. Then NeuEx adds these clauses and queries the SymSolv with these new symbolic constraints (Line 12). This method of adding conflict clauses is similar to the backtracking in DPLL algorithm [51]. Although the conflict clause learning approach used in NeuEx is simple, NeuEx can adopt other advance strategies for constraint solving in the future [75].

The above mixed constraint solving keeps executing the backtracking procedure until it does not find any new counterexample. Consider the example in Figure 1. NeuEx first

TABLE II. ENCODING SYMBOLIC CONSTRAINTS TO LOSS FUNCTIONS. a AND b REPRESENT ARBITRARY EXPRESSIONS. S_1 AND S_2 REPRESENT ARBITRARY SYMBOLIC CONSTRAINTS. L REPRESENTS THE LOSS FUNCTION USED FOR NEURAL CONSTRAINT SOLVING. L_{S_1} AND L_{S_2} REPRESENT THE LOSS FUNCTION FOR SYMBOLIC CONSTRAINTS S_1 AND S_2 RESPECTIVELY. α REPRESENTS A SMALL POSITIVE VALUE. β REPRESENTS A NON-ZERO SMALL REAL VALUE.

Symbolic Constraint	**Loss Function (L)**
$S_1 ::= a < b$	$L = max(a - b + \alpha, 0)$
$S_1 ::= a > b$	$L = max(b - a + \alpha, 0)$
$S_1 ::= a \leq b$	$L = max(a - b, 0)$
$S_1 ::= a \geq b$	$L = max(b - a, 0)$
$S_1 ::= a = b$	$L = abs(a - b)$
$S_1 ::= a \neq b$	$L = max(-1, -abs(a - b + \beta))$
$S_1 \wedge S_2$	$L = L_{S_1} + L_{S_2}$
$S_1 \vee S_2$	$L = min(L_{S_1}, L_{S_2})$

finds the values for variables `max`, `infilename`, and `infile` which satisfy Equations (1)-(3). If these values do not satisfy the neural constraint (Equation (4)), NEUEX transforms these values into a conflict clause and attempts a retrial to discover new values. However, the backtracking procedure may not terminate either because the constraints are complex or `UNSAT`. To avoid an infinite number of trials, NEUEX chooses to limit the number to a user-controlled threshold value. Specifically, if we do not have a `SAT` decision after mixed constraint solving I within $\texttt{MAX}_2$ iterations.[2]

NEUEX applies an alternative strategy where we combine the symbolic constraints with neural constraints together. There exist two possible strategies: transforming neural constraints into symbolic constraints or the other way around. However, collapsing neural constraints to symbolic constraints result in large sizes of encoded clauses. For example, encoding a small binarized neural network generates millions of variables and clauses [79]. Thus, we transform the mixed constraints into purely neural constraints for solving them together.

Mixed Constraint Solving II. The key idea for solving mixed constraints efficiently is to collapse symbolic constraints to neural constraints by encoding the symbolic constraints to a corresponding loss function (Line 28). This ensures the symbolic and neural constraints are in the same form. Table II shows the encoding of symbolic constraints and the loss function that NEUEX transforms it to. For Figure 6, NEUEX transforms the constraints S_2 and S_3 into a loss function N_2.

Once the symbolic constraints are encoded into neural constraints, NEUEX applies the NeuSolv to minimize the loss function on Line 30. The main intuition behind this approach is to guide the search with the help of encoded symbolic constraints. The loss function measures the distance between the current result and the satisfiable result of symbolic constraints. The search algorithm gives us a candidate value for satisfiability checking of neural constraints. However, the candidate value generated by minimizing the distance may not always satisfy the symbolic constraints since the search algorithm only tries to minimize the loss, rather than exactly forces the satisfiability of symbolic constraints. To weed out such cases, NEUEX checks the satisfiability for the symbolic constraints by plugging in the candidate value and querying the SymSolv on Line 31. If the result is `SAT`, the solver goes to `SAT` state. Otherwise, it continues executing Approach II with a different initial state of the search algorithm. For example, in Figure 6, NEUEX changes the initial value of V_7 for every iteration. Note that each iteration in Approach I has to execute sequentially because the addition of the conflict clause forces serialization. In contrast, each trial in Approach II is independent and thus embarrassingly parallelizable. To avoid the non-termination case, NEUEX sets the maximum number of trials for mixed constraint solving II to be $\texttt{MAX}_1$, which can be configured independently of our constraint solver.

Thus, neuro-symbolic execution has the ability to reason about purely symbolic constraints, purely neural constraints, and mixed neuro-symbolic constraints. This approach has a number of possible future applications, including but not limited to: (a) analyzing protocol implementations without analyzable code [49]; (b) analyzing code with complex dependency structures [97]; and (c) analyzing systems that embed neural networks directly as sub-components [36].

E. Encoding Mixed Constraints

One of the key challenges is in solving mixed constraints. To solve mixed constraints, we encode symbolic constraints as a loss function in the neural network. The variable values that minimize this loss function are expected to be close (ideally equal to) those which satisfy both the encoded symbolic constraints and the neural ones. Let X be the free input variables in the constraints and $S(X)$ be the symbolic constraints defined over a subset of X. We wish to define a loss function $L(X)$ such that the values of X that minimize $L(X)$ simultaneously satisfy $S(X)$. We define an encoding procedure for each kind of symbolic constraint into a corresponding loss function. NEUEX minimizes the joint loss functions of all symbolic constraints encoded as loss functions, together with that of the neural constraint. A gradient-based minimization procedure finds the minimum values of the joint loss function. The encoding and minimization procedure is explained next.

Encoding. For each kind of symbolic constraint in our language (Table I), we define a corresponding loss function. All string expressions are converted into bit-vectors [90] and treated like numeric variables (type `NumVar` in Table I). Table II describes the loss function for all six symbolic constraint types over numerics and the two constraint types over Booleans. Taking $a = b$ as an example, the loss function $L = abs(a - b)$ achieves the minimum value 0 when $a = b$, where a and b can be arbitrary expressions. Thus, minimizing the loss function L is equivalent to solving the symbolic constraint. We point out that encodings other than the ones we outline are possible. They can be plugged into NEUEX, as long as they adhere to the three requirements outlined next.

1) *Differentiability.* NeuSolv can only be applied to differentiable loss functions, as is the case with our encodings for each expression in Table II.
2) *Non-Zero Gradient Until* `SAT`. The derivative of the loss function should not be zero until we find the satisfiable assignments. For example, consider our encoding of the constraint $a < b$. Here, the derivative of the loss function should not be equal to zero when $a = b$. If this happens, NeuSolv will stop searching and return `UNSAT`. To avoid this, we add

[2]Users can adapt $\texttt{MAX}_2$ according to their applications.

a small positive value α, making the loss function $L = max(a - b + \alpha, 0)$. The cases for $a > b$ and $a \neq b$ are similar to the constraint $a < b$.

3) *Finite Lower Bound for Loss Functions.* The loss function for each constraint needs to have a finite lower bound. Without this, the procedure would continue minimizing one of the constraints indefinitely in (say) a conjunction of clauses. For instance, our encoding of $a \neq b$ as $L = max(-1, -abs(a-b+\beta))$ carefully ensures a finite global minimum.[3] If we instead encoded it as $L = -abs(a - b + \beta)$, the loss function would have no finite minimum. When we consider the conjunction of two clauses, say $(a \neq b) \wedge (c < d)$, the joint loss function for the conjunction of the two clauses is the sum of the individual losses. NeuSolv may not know where to terminate the minimization of the loss for $(a \neq b)$, preventing it from finding the satisfiable assignment for the conjunction. To avoid this, our encoding adds an explicit lower bound of -1.

F. Optimizations

NEUEX applies five optimization strategies to reduce the computation time for neuro-symbolic constraint solving.

Single Variable Update. Given a set of input variables to neural constraint, NEUEX only updates one variable for each enumeration in NeuSolv. In order to select the variable, NEUEX computes the derivative values for each variable and sorts the absolute values of derivatives. The updated variable is the one with the largest absolute value of the derivative. This is because the derivative value for each element only computes the influence of changing the value of one variable towards the value of loss function, but does not measure the joint influence of multiple variables. Thus, updating them simultaneously may increase the loss value. Moreover, updating one variable per iteration allows the search engine to perform the minimum number of mutations on the initial input in order to prevent the input from being invalid.

Type-based Update. To ensure the input is valid, NEUEX adapts the update strategy according to the types of variables. If the variable is an integer, NEUEX first binarizes the value of derivatives and then updates the variables. If the variable is a float, NEUEX updates the variable with the actual derivatives.

Caching. NEUEX stores the updated results for each enumeration in NeuSolv. As the search algorithm is a deterministic approach, the final generated result is the same if we have the same input, neural constraints, and the loss function. Thus, to avoid unnecessary re-computation, NEUEX stores the update history and checks whether current input is cached in history. If yes, NEUEX reuses the previous result; otherwise, NEUEX keeping searching for a new input.

SAT Checking per Enumeration. To speed up the solving procedure, NEUEX verifies the satisfiability of the variables after each enumeration in NeuSolv. Once it satisfies the symbolic constraints, NeuSolv terminates and returns `SAT` to NEUEX. This is because not only the result achieving global minima can be the satisfiable result of symbolic constraint. For example, any result can be the satisfiable result of the constraint $a \neq b$ except for the result satisfying $a = b$. Hence, NEUEX does not wait for minimizing the loss function, but checks the updated result for every iteration.

Parallelization. NEUEX executes NeuSolv with different initial input in parallel since each loop for solving mixed constraints is independent. This parallelization reduces the time for finding the global minima of the loss function.

V. IMPLEMENTATION

We implement NEUEX in KLEE v1.4. We use KLEE-uClibc [28] and Z3 SMT solver [53] as the back-end for KLEE. Our implementation makes 351 LOC change to KLEE for monitoring and passing run-time information to NEUEX. We build our static analysis using Clang v3.4 and Clang Static Analyzer [6] in 601 LOC. Rest of NEUEX is implemented in Python and TensorFlow with 4635 LOC.

Profiler. We launch KLEE with its logging turned on, such that it reports external calls, SMT query execution time, loop unrolling, and increase in memory footprints due to path explosion. Further, we also ask KLEE to log the instructions that it symbolically executes. We directly use the appropriate flags provided by KLEE to turn on this logging. NEUEX's profiler continuously monitors these logs as they are being written out. We coarsely detect loop unrolls via scanning the instruction log i.e., when line numbers in the program code are being periodically. Our profiler starts the neural mode process whenever it detects the following events: (a) warnings for external calls; (b) Z3 threshold limit capped at 10 minutes; (c) loop unroll count is greater than $10,000$ (d) state termination because of memory cap (3 GB). NEUEX launches *a separate neural-mode process* which has an RPC tunnel to KLEE. It then signals KLEE to pass the current program point current symbolic state, the inputs, and the symbolic path constraints it has collected so far to this newly created neural-mode process.

CVP Reachability. NEUEX reasons about an arbitrary size program code by representing it as a neural net. When NEUEX starts its neural mode it takes the symbolic states from KLEE and sends a signal to KLEE to abort these states to continue on a different path. We continue in neural mode from this symbolic state onwards to multiple CVPs in the rest of the program. Specifically, we select the closest k (default 150) CVPs for each bug type in the static call graph. For each CVP, we concretize all the symbolic states and create a random seed for input generation. Specifically, we generate $20,000$ random inputs and execute the program with these concrete inputs. When NEUEX finds at least one input which reaches the CVP of our choice, we consider that the CVP is reachable.

Sample Set Generation. We generate more inputs by using the reachable input created in the previous step as the seed. We randomly mutate the seed input to produce new inputs and queue them in a working set. We persist this working set in form of files for ease of use, thus freeing up memory. NEUEX then spawns multiple new target programs with the inputs from the working set in batches of $10,000$. At the end of each execution, NEUEX logs the values of variables of interest

[3]Here β can be any non-zero and small real value.

TABLE III. NUMBER OF VERIFIED BUGS FOUND BY NEUEX VS. VANILLA KLEE IN A 12 HOUR RUN, EACH CONFIGURED WITH BFS AND RAND MODE IN SEPARATE EXPERIMENTS. THE "COMBINED" COLUMN REPORTS THE TOTAL EXPLOITS FOUND IN EITHER MODE.

Program	Known CVEs	Vanilla KLEE			NEUEX		
		BFS	Random	Combined	BFS	Random	Combined
cURL	[19]	1	2	2	1	2	2
SQLite	[18]	0	0	0	2	2	2
libTIFF	[17], [23]–[25]	0	0	0	4	4	4
libsndfile	[20]–[22]	0	0	0	3	3	3
BIND	[3], [7], [13]	1	1	1	5	5	5
Sendmail	[4], [5], [8]–[10], [14], [15]	11	11	11	12	12	12
WuFTP	[11], [12], [16]	4	4	4	7	7	7
Total		**17**	**18**	**18**	**33**	**34**	**34**
No. of Unknown Exploits		8	9	9	11	12	12

at various CVPs as entries in the respective CVP's sample set. Our sample sets are in the form of input-output files per entry, so we have configured all our programs to take file-based input and produce file based outputs. We use unique file names for each execution. We can scale this process to multiple cores and/or physical machines since each execution is independent. We implement our generator in Python with 656 LOC.

Side-effects. In our experiments, none of the programs read or modify any global states of the machine environment (e.g., configuration files). They only take in one input file and optionally produce an output file or write to the console. Thus it is safe to execute the same program multiple times with unique input file names and redirect the output to different files. In cases where this does not hold true, NEUEX piggybacks on the environment modeling of KLEE. Specifically, KLEE models a simple symbolic file system. It redirects all the environment related calls to stubs which model the behavior of these file APIs. NEUEX hooks these stubs and in cases where the data set generation may affect a global state on a write, we instead redirect such calls to virtual files in memory with locks for avoiding global state corruption. This way, we can isolate global changes made by each execution. Further, our sample set generation is a different process, so it does not interfere with the execution of DSE mode of KLEE.

Training. Next, the neural constraint inference engine takes 80% of all the generated sample sets for training the neural net. We use a standard MLP architecture with `Relu` activation function implemented in Python and Google TensorFlow [29] with a total of 208 LOC. We use the early-stopping strategy to avoid over-fitting. We test the remaining 20% of the sample set on the learned neural net to measure its accuracy. We continue the training until we achieve at least 80% accuracy. If the loss of the trained neural net does not start decreasing after a threshold of 50 epochs, we discard the search for an exploit for the corresponding CVP.

Solver. Finally, NEUEX solves the learned neural constraints along with the symbolic constraints collected from KLEE. We implement our Algorithm 1, symbolic constraint transformation (Table II), and a standard gradient-based optimization algorithm in Python with 849 LOC. Our implementation optionally takes into consideration the type of the input variables if it is easily available from the source code. This auxiliary information helps us to select the step increment size, thus accelerating the search for the exploit. Specifically, the step size is integer value and floating-point value for integer and real data-types, respectively. After each enumeration, we execute the program with concrete outputs generated by our solver to check if they indeed satisfy the neuro-symbolic constraints.

Parallelization. We have described all the steps for the end-to-end working of NEUEX. NEUEX can execute these steps sequentially or use parallelization to speed up certain tasks. We configure NEUEX to execute on n cores. We dedicate one core for KLEE's classic DSE mode which is inherently serial, and leverage the rest of the $n-1$ cores for neural mode. Our reachability check for CVPs, sample set generation, and training steps are individually independent and can execute on multiple cores. The inference step is strictly sequential for each initial input, however, we can execute the solver with different initial inputs in parallel. Further, processing for multiple CVPs can be pipelined such that NEUEX starts processing the next CVP while finishing the sequential steps of the previous CVP.

VI. EVALUATION

We show the effectiveness of NEUEX by answering the following empirical questions:

- *Efficiency.* Does NEUEX improve over vanilla KLEE?
- *Cost breakdown of neural mode.* How many times is the neural mode of NEUEX triggered? What are the relative costs of various sub-steps in NEUEX?
- *Comparison with structured constraint inference tools.* How does the neural mode of NEUEX compare to the structured inference approaches which augment dynamic symbolic execution?

Experimental Setup. All our experiments are performed on a 40-core 2.6GHz 64GB RAM Intel Xeon machine. KLEE uses a single core in its operations by design [40], whereas NEUEX is highly parallelizable. We set thresholds for terminating DSE to 10 minutes for Z3 per constraint, memory to 3 GB, loop count to $10,000$, and the maximum enumeration of NeuSolv M_e to $50,000$. We avoid duplicates in counting vulnerabilities by using the unique stack hash at the point of crash [78]. We further verify that the input indeed triggers an exploit by re-running the test files generated by NEUEX.[4]

Benchmarks. For evaluation, we select 7 programs reported in Table III. Since our comparison is with DSE as a baseline, we choose programs that are known to be difficult for it

[4]KLEE reports a total of 34 bugs, out of which only 17 are unique and true exploits. Rest of them are either duplicates or false-positives due to KLEE's imprecise internal modeling of the concrete memory.

TABLE IV. NEUEX NEURAL MODE PERFORMANCE BREAKDOWN. DETAILED STATISTICS OF VULNERABILITIES DISCOVERED BY THE NEURAL MODE OF NEUEX IN 12 HOURS.

Program	#Times Neural Mode Triggered	CVPs Covered in Neural Mode	#CVPs with Sufficient Training Dataset	Successfully Learned Networks	Verified Exploits	Time (hour)
cURL	1	6	2	1	1	7
SQLite	5	24	12	3	1	0.7
libTIFF	7	6	2	1	1	5.8
	2	14	5	2	1	7.6
	6	17	7	1	1	11.9
	4	9	4	1	1	2.7
libsndfile	5	14	4	4	1	2
	5	14	4	4	1	1.3
	2	4	1	1	1	8.5
BIND	1	7	5	1	1	0.4
	11	13	4	1	1	3
	1	3	2	1	1	1
	1	5	3	1	1	1.2
Sendmail	1	5	3	1	1	0.1
	1	5	1	1	1	0.2
	2	6	1	1	1	0.3
	3	3	1	1	1	0.2
WuFTP	3	4	3	3	3	0.4
Total	**61**	**159**	**64**	**29**	**20**	**-**

due to complex loops, floating-point variables, and unknown function calls. 3 of these programs comprise a standard benchmark (LESE [101]) used in prior work that improves loop-handling over DSE, and the remaining are real-world applications for media processing, web data transfer, and database management. All the programs have prior publicly known vulnerabilities: various out-of-bound accesses, floating point exceptions, and arithmetic overflows. We manually craft input grammars for these programs and use the same symbolic inputs for KLEE and NEUEX.

A. *Efficiency of* NEUEX *over KLEE*

We run vanilla KLEE and NEUEX on each of the programs with a 12-hour timeout. We experimented with two search strategies in KLEE: (a) `BFS` mode, where symbolic states to solve are picked in breadth-first search order of the control-flow graph, and (b) the KLEE default `RAND` mode, where paths are selected randomly favoring new paths. Since NEUEX builds on KLEE, we ran NEUEX with both these strategies.

As shown in Table III, within 12 hours, vanilla KLEE finds 17 and 18 bugs in `BFS` and `RAND` mode respectively. NEUEX finds all bugs that vanilla KLEE does, and additionally finds 16 more, totaling to 33 and 34 bugs in `BFS` and in `RAND` mode respectively. Out of the bugs found by vanilla KLEE and also by NEUEX, 12 bugs are previously unknown, whereas all bugs found by NEUEX alone are previously known CVEs. All the 12 previously unknown vulnerabilities found are out-of-bound buffer accesses: 1 in Bind, 1 in cURL, 6 in Sendmail, and 4 in WuFTP. We have responsibly disclosed these bugs to the corresponding application maintainers.

NEUEX finds 94% and 89% more bugs than vanilla KLEE in `BFS` and `RAND` modes respectively, within the same time window. To explain the improvements, we further compare KLEE and NEUEX in the `BFS` mode, which has deterministic exploration strategy and is hence unaffected by the randomness internally used by the tools. Figure 7 shows that the number of CVPs reached or covered by NEUEX is significantly higher than vanilla KLEE. NEUEX covers these CVPs much faster. Recall that the neural mode helps classic DSE mode in NEUEX in two conceptual ways. First, it drives down paths which are

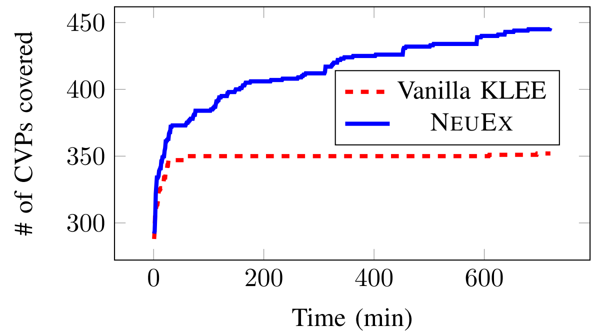

Fig. 7. CVP Coverage. Each line represents the number of CVPs covered or reached by vanilla KLEE and NeuEx in `BFS` mode, as the function of time. The solid line represents the CVPs covered by NEUEX. The dashed line represents the CVPs covered by vanilla KLEE.

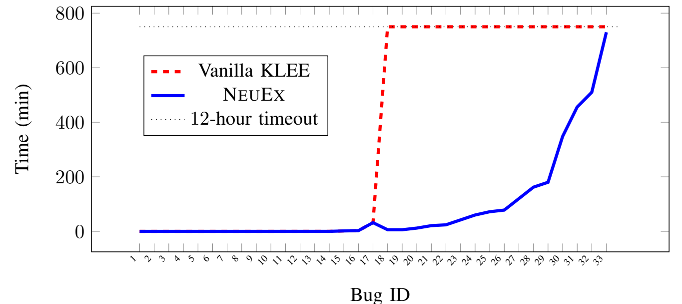

Fig. 8. Exploit Generation Time. The time taken to generate exploits (y-axis) for each bug ID (x-axis) in `BFS` mode by vanilla KLEE and NEUEX. The solid line represents the time taken by NEUEX. The dashed line represents the time taken by vanilla KLEE. The dotted line represents the 12-hour timeout. KLEE is not able to find the bugs which reach the 12-hour timeout.

otherwise difficult and which cause vanilla DSE (and KLEE) to gets stuck. Second, the classic DSE mode is terminated on states where it gets stuck, allowing it to explore other easier paths rather than spending time on difficult ones. Intuitively, terminating DSE mode on difficult paths can itself enhance coverage of NEUEX by allowing DSE to cover other paths. We experimentally check if this holds true. We find that without the neural reasoning, NEUEX performs marginally better than vanilla KLEE with 2% more coverage and no additional exploits. This shows that the neural reasoning in NEUEX leads to its enhanced coverage.

Column 2 of Table IV reports that the neural mode is triggered 61 times in the classical DSE mode. These triggers are due to 6 external calls without stubs, 53 loop timeouts, 1 timeouts of the Z3 solver, and 1 memory exhaustion instances in our experiments, directly highlighting the bottlenecks of KLEE which NEUEX resolves.

Figure 8 reports the total time taken to find different exploits by NEUEX and vanilla KLEE in its `BFS` mode in log-scale (base 2). The 17 vulnerabilities found by both tools are found relatively quickly. Though 4 of them trigger the neural mode, the `BFS` procedure reaches them faster than the neural mode, by brute-forcing the path space. 16 out of 33 found by NEUEX are not found by vanilla KLEE even in 12 hours, however, showing the advantage of the neural mode. Our subsequent manual analysis confirms the underlying reason:

TABLE V. COMPARISON RESULTS BETWEEN THE NEURAL MODE OF NEUEX AND LESE ON FINDING THE SAME VULNERABILITIES ON THE BENCHMARK WITH THE SAME SETUP AND SMALLER SAMPLE SET SIZE.

Program	**BIND**				**Sendmail**							**WuFTP**			**Geometric Mean**
Vulnerability	**CA-1999-14(1)**	**CA-1999-14(2)**	**CVE-1999-0009**	**CVE-2001-0013**	**CA-2003-07**	**CVE-1999-0131**	**CVE-1999-0206**	**CVE-1999-0047**	**CA-2003-12**	**CVE-2001-0653**	**CVE-2002-0906**	**CVE-1999-0878**	**CAN-2003-0466**	**CVE-1999-0368**	
LESE (s)	2511	2155	586	4464	672	526	626	633	18080	676	237	483	197	109	**2282.5**
Neural Mode of NEUEX (s)	30.46	25.48	37.65	14.34	6.13	3.53	4.31	6.22	4.32	23.71	144.41	13.89	10.10	7.20	**23.70**
Speedup Factor	82.44 ×	84.58 ×	15.57 ×	311.30 ×	109.62 ×	149.01 ×	145.24 ×	101.77 ×	4185.19 ×	28.51 ×	1.64 ×	34.77 ×	19.50 ×	15.14 ×	**377.45** ×

on simple constraints, such as linear relationships, both vanilla KLEE and neural analysis work relatively fast, with vanilla KLEE being faster as it captures the constraint symbolically. As constraints become complex (see example in Section II), vanilla KLEE becomes considerably slower and cannot recover relationships within 12 hours in the extreme cases. This shows that the neural mode gracefully scales with the increasing complexity of constraints to be recovered.

B. NEUEX *Performance Breakdown*

When neural mode is triggered, NEUEX spends time in 4 sub-steps: (a) trying to reach certain CVPs; (b) generating I/O training value to train each reached CVP; (c) training one neural net for each CVP; and (d) solving the mixed neuro-symbolic constraint to generate exploit inputs. Note that in step (d), exploits are enumerated up to a maximum of $50,000$ and NEUEX stops searching when one working exploit (that is concretely validated) is found for a CVP. Table IV reports the number of CVPs for each of the categories (a)-(d) solely due to its neural mode, and the total time to find the verified exploit. Neural mode is triggered 61 times, leading to 20 exploits, out of which 16 are only found by the neural mode.[5]

Breakdown by CVPs. Table IV reports the number of CVPs which are reached / covered by NEUEX (Column 3) in neural mode. For a fraction (64 out of 159) of the covered CVPs, NEUEX is able to generate sufficient data samples to train the neural network (Column 4). Note that NEUEX generates random state mutations from the state that the DSE mode gets stuck in order to reach target CVPs, hence not all CVPs are reached in our experiments. Out of those 64 with sufficient training data samples, NEUEX is able to successfully learn neural networks for 29 CVPs (Column 5); here, we consider networks which achieve an accuracy of at least 50% as successfully learnt. We use the standard accuracy metric.[6] The number of samples to train one neural net successfully is less than $200,000$. Out of the successfully trained neural networks, NEUEX uses its neuro-symbolic solver to generate inputs which exploit the CVP. We find that 20 of the 29 successfully learnt neural networks lead to verified (or true) exploits, which are reported to the end user.

CPU Cycle Breakdown. Figure 9 shows the fraction of CPU cycles NEUEX spends in sub-steps (a)-(d) for each program. The majority of time spent is on dataset generation for training neural network, followed by the training itself. The dataset

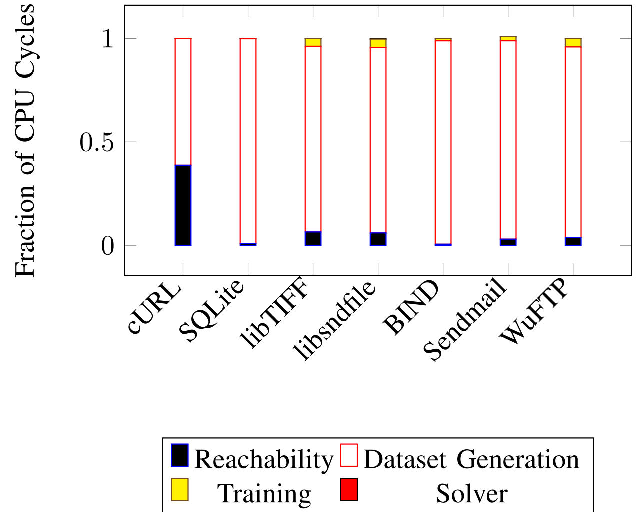

Fig. 9. CPU Cycle Breakdown for NEUEX Neural Mode. The size of each shaded region represents the fraction of cycles NEUEX takes for each step.

generation cost is impacted by the size of the program fragment being approximated by the neural network—larger fragments take more CPU cycles to execute. Our implementation takes advantage of parallel cores for sub-steps (a) and (b), specifically the mutations for reaching CVPs and dataset generation for each CVP reached respectively. The training and solving for each CVP runs on 1 core each. The neural constraint solving is fast for most cases.

Benefits of Automatically Learning Stubs. Through manual analysis, we find that the ability of NEUEX to approximate missing code with automatically learnt stubs leads to several CVEs. For example, in `libsndfile`, NEUEX automatically approximate the `fabs` function when learning the relationship between input bytes and the vulnerable variable `max`. Due to the precise approximation, NEUEX takes 1.3 hour to successfully generate the exploits triggering the vulnerability. Similarly, NEUEX provides sufficient stubs for function `dn_skipname` in `bind` utility for successfully generating the exploits. Further, it can reach the vulnerability point in `libTIFF` library despite the missing `mmap` function stub.

C. Comparison to Structured Constraint Inference

Learning neural constraints is faster for simple (e.g., linear) relationships, but with additional time, much more complex relations can be learnt. To show this, we compare to an extension of DSE called LESE that learns specific linear relationships. NEUEX is faster than LESE by two orders of magnitude as shown in Table V. We explain conceptually why

[5]The rest of the bugs found by NEUEX are found in its classical DSE mode and by vanilla KLEE too.

[6]$Accuracy = 100 \times \frac{\#\ of\ correctly\ predicted\ samples}{\#\ of\ test\ samples}$. The generated sample dataset is partitioned into disjoint sets for testing (20%) and training (80%).

multi-path extensions to DSE that ameliorate the low-level path explosion do *not* improve their ability to recover succinct representations in Section II. We experimentally confirm this by testing state-of-the-art techniques called LESE [91] and Veritesting [33]. Both these experiments are detailed here.

LESE. LESE augments symbolic execution for reasoning about loops on binaries [91]. It learns linear relationships between loop induction variables. As the source code of LESE is not public, in this experiment we evaluate the neural mode of NEUEX on the same benchmark that LESE uses on an identical setup to compare the numbers reported in their paper. The LESE benchmark consists of 3 programs with 14 bugs in total [101]. We task the neural mode of NEUEX to find exploits for these 14 vulnerabilities. We configure NEUEX to execute only in neural mode because we want to compare LESE technique directly to our neuro-symbolic execution. Table V summarizes the execution time for NEUEX's neural mode and LESE to complete the benchmark. NEUEX finds the exploits for all the 14 bugs in the benchmark under 3 minutes, while LESE takes 5 hours in the worst case. NEUEX's neural mode is faster than LESE by two orders of magnitude on average. Thus, our comparison to LESE demonstrates that NEUEX's neural mode can handle loops by synthesizing the loop constraints in form of neural net without analyzing the program.

Veritesting. We compare NEUEX with Veritesting [33]. Since the original implementation used in the paper is not publicly available, we instead used its publicly available re-implementation as part of the Angr binary analysis framework [1], [94]. We report on the cURL program for comparison as a starting point, because in our experiments vanilla KLEE was able to find the cURL exploit in 34 minutes, so a conceptual advance on KLEE should be able to identify it faster. The Veritesting implementation on Angr requires a target address that it aims to find exploits at. We provided the known vulnerability point address in cURL as a target address to this implementation. We gave the same symbolic arguments for Angr as we did for KLEE and NEUEX. We then used Angr's symbolic execution routine with the Veritesting flag turned on to find concrete inputs to reach the vulnerability. In 12 hours, it analyzed a total of 530 unique instructions, $2,353$ symbolic states, 38 static analysis calls for Veritesting and 198 unique external functions where stubs were not available. Veritesting was not able to reach the vulnerability in 12 hours.

VII. RELATED WORK

NEUEX is a new design point in constraint synthesis and constraint solving. In this section, we discuss the problems of the existing symbolic execution tools and present how NEUEX differs from existing constraint synthesis.

A. Symbolic Execution

Symbolic execution [68] has been used for program verification [50], software testing [40], and program repair via specification inference [80]. In the last decade, we have witnessed an increased adoption of dynamic symbolic execution [62] where symbolic execution is used to partition the input space, with the goal of achieving increased behavioral coverage. The input partitions computed are often defined as program paths, all inputs tracing the same path belong to the same partition. Thus, the test generation achieved by dynamic symbolic execution suffers from the path explosion problem. This problem can be exacerbated owing to the presence of complex control flows, including long-running loops (which may affect the scalability of dynamic symbolic execution since it involves loop unrolling) and external libraries. However, NEUEX does not suffer from the path explosion as it learns the constraints from test executions directly.

Tackling path explosion is a major challenge in symbolic execution. Boonstopel et al. suggest the pruning of redundant paths during the symbolic execution tree construction [37]. One of the predominant ways of tackling the path explosion problem is by summarizing the behavior of code fragments in a program [31], [33], [60], [72], [92]. Simply speaking, a summarization technique provides an approximation of the behavior of certain fragments of a program to keep the scalability of symbolic execution manageable. Such an approximation of behaviors is also useful when certain code fragments, such as remote calls and libraries written in a different language, are not available for analysis.

Among the past approaches supporting approximation of behaviors of (parts of) a program, the use of function summaries has been studied by Godefroid [60]. Such function summaries can also be computed on-demand [31]. Kuznetsov et al. present a selective technique to merge dynamic states. It merges two dynamic symbolic execution runs based on an estimation of the difficulty in solving the resultant Satisfiability Modulo Theory (SMT) constraints [72]. Veritesting suggests supporting dynamic symbolic execution with static symbolic execution thereby alleviating path explosion due to factors such as loop unrolling [33], which still suffers from unknown function calls and SMT solver timeouts. We conceptually and experimentally compare to this approach. Related works [84], [92] suggest grouping together paths based on similar symbolic expressions in variables, and use such symbolic expressions as dynamic summaries to group paths.

B. Constraint Synthesis

To support the summarization of program behaviors, the other core technical primitive we can use is constraint synthesis. In our work, we propose a new constraint synthesis approach which utilizes neural networks to learn the constraints which are infeasible for symbolic execution. In comparison with previous solutions, the major difference is that NEUEX does not require any pre-defined templates of constraints and can learn any kind of relationships between variables.

Over the last decade, there are two lines of works in constraint synthesis: white-box and black-box approaches. White-box constraint inference relies on a combination of light-weight techniques such as abstract interpretation [47], [48], [86], interpolation [43], [67], [76] or model checking algorithm IC3 [38]. Although some white-box approaches can provide sound and complete constraints [46], it is dependent on the availability of source code and a human-specified semantics of the source language. Constructing these tools have required considerable manual expertise to achieve precision, and many of these techniques can be highly computationally intensive.

To handle the unavailability of source code, there also exist a rich class of works on reverse engineering from

dynamic executions [54], [58], [66], [81]–[83]. Such works can be used to generate summaries of observed behavior from test executions. These summaries are not guaranteed to be complete. On the other hand, such incomplete summaries can be obtained from tests, and hence the source code of the code fragment being summarized need not be available. Daikon [54] is one of the earlier works proposing a synthesis of potential invariants from values observed in test executions. The invariants supported in Daikon are in the form of linear relations among program variables. DIG extends Daikon to enable dynamic discovery of non-linear polynomial invariants via a combination of techniques including equation solving and polyhedral reasoning [82]. Krishna et al. use the decision tree, a machine learning technique, to learn the inductive constraints from good and bad test executions [70].

NEUEX devises a new gradient-based constraint solver, the first work which solves the conjunction of neural and SMT constraints. Angora [42] uses gradient-based approach which is similar to NEUEX, albeit for a completely different usage. It treats the predicates of branches as a black-box function which is not differentiable, while NEUEX encodes the symbolic constraints into a differentiable function and embeds it into neural constraints. A concurrent work, NeuZZ [93], utilizes neural networks to guide random fuzzing to predict the control-flow edges exercised by a given input. Our work instead uses neural networks to improve upon dynamic symbolic execution. Li et al. [74] propose a method to solve symbolic constraints using a classification-based optimization technique called RACOS [99], instead of using SAT/SMT solvers. Unlike our work, their work does not attempt to learn a non-symbolic representation of the program to ameliorate the difficulties of symbolic analyses, and hence our work proposes entirely different constraint solving techniques.

VIII. CONCLUSION

NEUEX utilizes neural networks to inductively learn constraints which approximate program behavior. Our proposed neuro-symbolic execution solves neural and symbolic constraints together, and can be seen as a general purpose testing and analysis engine for programs. NEUEX's solver offers a new design to simultaneously solve both symbolic constraints and neural constraints effectively, thus augmenting symbolic execution. Our technique finds 94% more bugs than vanilla dynamic symbolic execution.

ACKNOWLEDGMENTS

We thank Marcel Böhme, Shruti Tople, Shin Hwei Tan, Xiang Gao, Sergey Mechtaev, the anonymous reviewers, and our shepherd Endadul Hoque for their feedback on this work. We thank Changze Cui for helping us in the most recent version of our implementation and experiments. Thanks to Vinamra Bhatia for helping on Veritesting. All opinions expressed in this paper are solely those of the authors. This research is supported by research grant DSOCL17019 from DSO, Singapore. This research was partially supported by a grant from the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (TSUNAMi project, No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate.

REFERENCES

[1] Angr. https://github.com/angr/angr.

[2] BIND DNS Server. https://www.isc.org/downloads/bind/.

[3] CA-1999-14. https://www-uxsup.csx.cam.ac.uk/pub/webmirrors/www.cert.org/advisories/CA-1999-14.html.

[4] CA-2003-07. https://lwn.net/Articles/24238/.

[5] CA-2003-12. https://seclists.org/cert/2003/12.

[6] Clang Static Analyzer. https://clang-analyzer.llvm.org/.

[7] CVE-1999-0009. https://nvd.nist.gov/vuln/detail/CVE-1999-0009.

[8] CVE-1999-0047. https://nvd.nist.gov/vuln/detail/CVE-1999-0047.

[9] CVE-1999-0131. https://nvd.nist.gov/vuln/detail/CVE-1999-0131.

[10] CVE-1999-0206. https://nvd.nist.gov/vuln/detail/CVE-1999-0206.

[11] CVE-1999-0368. https://nvd.nist.gov/vuln/detail/CVE-1999-0368.

[12] CVE-1999-0878. https://nvd.nist.gov/vuln/detail/CVE-1999-0878.

[13] CVE-2001-0013. https://nvd.nist.gov/vuln/detail/CVE-2001-0013.

[14] CVE-2001-0653. https://nvd.nist.gov/vuln/detail/CVE-2001-0653.

[15] CVE-2002-0906. https://nvd.nist.gov/vuln/detail/CVE-2002-0906.

[16] CVE-2003-0466. https://nvd.nist.gov/vuln/detail/CVE-2003-0466.

[17] CVE-2014-8130. https://nvd.nist.gov/vuln/detail/CVE-2014-8130.

[18] CVE-2015-3416. https://nvd.nist.gov/vuln/detail/CVE-2015-3416.

[19] CVE-2016-9586. https://nvd.nist.gov/vuln/detail/CVE-2016-9586.

[20] CVE-2017-14245. https://nvd.nist.gov/vuln/detail/CVE-2017-14245.

[21] CVE-2017-14246. https://nvd.nist.gov/vuln/detail/CVE-2017-14246.

[22] CVE-2017-16942. https://nvd.nist.gov/vuln/detail/CVE-2017-16942.

[23] CVE-2017-7598. https://nvd.nist.gov/vuln/detail/CVE-2017-7598.

[24] CVE-2017-7599. https://nvd.nist.gov/vuln/detail/CVE-2017-7599.

[25] CVE-2017-7600. https://nvd.nist.gov/vuln/detail/CVE-2017-7600.

[26] Klee uClibc. https://github.com/klee/klee-uclibc.

[27] STP – The Simple Theorem Prover. https://stp.github.io/.

[28] uClibc. https://www.uclibc.org/.

[29] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "TensorFlow: A System for Large-Scale Machine Learning," in *OSDI'16*.

[30] E. Ábrahám, "Building Bridges between Symbolic Computation and Satisfiability Checking," in *ISSAC'15*.

[31] S. Anand, P. Godefroid, and N. Tillman, "Demand-Driven Compositional Symbolic Execution," in *TACAS'08*.

[32] A. Andoni, R. Panigrahy, G. Valiant, and L. Zhang, "Learning Polynomials with Neural Networks," in *ICML'14*.

[33] T. Avgerinos, A. Rebert, S. Cha, and D. Brumley, "Enhancing Symbolic Execution with Veritesting," in *ICSE'14*.

[34] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A Survey of Symbolic Execution Techniques," *ACM Computer Survey'18*.

[35] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing Program Input Grammars," in *PLDI'17*.

[36] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang *et al.*, "End-to-End Deep Learning for Self-Driving Cars," *arXiv*, 2016.

[37] P. Boonstoppel, C. Cadar, and D. Engler, "RWset: Attacking Path Explosion in Constraint-Based Test Generation," in *TACAS'08*.

[38] A. R. Bradley, "SAT-Based Model Checking Without Unrolling," in *VMCAI'11*.

[39] A. Bundy and L. Wallen, "Breadth-first Search," in *Catalogue of Artificial Intelligence Tools*, 1984.

[40] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *OSDI'08*.

[41] C. Cadar and K. Sen, "Symbolic Execution for Software Testing: Three Decades Later," *Comm of ACM'13*.

[42] P. Chen and H. Chen, "Angora: Efficient Fuzzing by Principled Search," *SP'18*.

[43] Y.-F. Chen, C.-D. Hong, B.-Y. Wang, and L. Zhang, “Counterexample-Guided Polynomial Loop Invariant Generation by Lagrange Interpolation,” in *CAV’15*.

[44] V. Chipounov, V. Kuznetsov, and G. Candea, “S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems,” *ACM SIGPLAN Notices*, 2011.

[45] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezzé, “Using Symbolic Execution for Verifying Safety-Critical Systems,” in *ACM SE Notes’01*.

[46] M. A. Colón, S. Sankaranarayanan, and H. B. Sipma, “Linear Invariant Generation Using Non-Linear Constraint Solving,” in *CAV’03*.

[47] P. Cousot and R. Cousot, “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints,” in *POPL’77*.

[48] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, “The ASTRÉE Analyzer,” in *ESOP’05*.

[49] W. Cui, J. Kannan, and H. J. Wang, “Discoverer: Automatic Protocol Reverse Engineering from Network Traces,” in *USENIX Security’07*.

[50] R. Dannenberg and G. Ernst, “Formal Program Verification using Symbolic Execution,” in *IEEE TSE’82*.

[51] M. Davis, G. Logemann, and D. Loveland, “A Machine Program for Theorem-proving,” *Communications of the ACM*, 1962.

[52] E. de Castro Lopo. libsndfile. http://www.mega-nerd.com/libsndfile/.

[53] L. de Moura and N. Bjørner, “Z3: An Efficient SMT Solver,” in *TACAS’08*.

[54] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, “The Daikon system for dynamic detection of likely invariants,” in *Science of Computer Programming*, 2007.

[55] K.-I. Funahashi, “On the approximate realization of continuous mappings by neural networks,” in *Neural Networks’89*.

[56] V. Ganesh, A. Kieżun, S. Artzi, P. J. Guo, P. Hooimeijer, and M. Ernst, “HAMPI: A String Solver for Testing, Analysis and Vulnerability Detection,” in *CAV’11*.

[57] S. Gao, S. Kong, and E. M. Clarke, “dReal: An SMT Solver for Nonlinear Theories over the Reals,” in *CADE’13*.

[58] P. Garg, C. Löding, P. Madhusudan, and D. Neider, “ICE: A Robust Framework for Learning Invariants,” in *CAV’14*.

[59] X. Glorot, A. Bordes, and Y. Bengio, “Deep Sparse Rectifier Neural Networks,” in *AISTATS’11*.

[60] P. Godefroid, “Compositional Dynamic Test Generation,” in *POPL’07*.

[61] P. Godefroid, A. Kiezun, and M. Y. Levin, “Grammar-based Whitebox Fuzzing,” in *PLDI’08*.

[62] P. Godefroid, N. Klarlund, and K. Sen, “DART: Directed Automated Random Testing,” in *PLDI’05*.

[63] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, “Automated Whitebox Fuzz Testing,” in *NDSS’08*.

[64] L. B. Godfrey and M. S. Gashler, “A Continuum among Logarithmic, Linear, and Exponential Functions, and Its Potential to Improve Generalization in Neural Networks,” in *IC3K’15*.

[65] I. J. Goodfellow, J. Shlens, and C. Szegedy, “Explaining and Harnessing Adversarial Examples,” in *ICLR’15*.

[66] A. Gupta and A. Rybalchenko, “InvGen: An Efficient Invariant Generator,” in *CAV’09*.

[67] R. Jhala and K. L. McMillan, “A Practical and Complete Approach to Predicate Refinement,” in *TACAS’06*.

[68] J. C. King, “Symbolic Execution and Program Testing,” *Communications of the ACM*, 1976.

[69] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” in *ICLR’15*.

[70] S. Krishna, C. Puhrsch, and T. Wies, “Learning Invariants using Decision Trees,” *arXiv*, 2015.

[71] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” in *NIPS’12*.

[72] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, “Efficient State Merging in Symbolic Execution,” in *PLDI’12*.

[73] G. Li, E. Andreasen, and I. Ghosh, “SymJS: Automatic Symbolic Testing of JavaScript Web Applications,” in *FSE’14*.

[74] X. Li, Y. Liang, H. Qian, Y.-Q. Hu, L. Bu, Y. Yu, X. Chen, and X. Li, “Symbolic Execution of Complex Program Driven by. Machine Learning Based Constraint Solving,” in *ASE’16*.

[75] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, “Exponential Recency Weighted Average Branching Heuristic for SAT Solvers,” in *AAAI’16*.

[76] K. McMillan, “Interpolation and SAT-based Model Checking,” in *CAV’03*.

[77] L. Medsker and L. Jain, *Recurrent Neural Networks: Design and Applications*. CRC press, 1999.

[78] D. Molnar, X. C. Li, and D. Wagner, “Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs,” in *USENIX Security’09*.

[79] N. Narodytska, S. P. Kasiviswanathan, L. Ryzhyk, M. Sagiv, and T. Walsh, “Verifying Properties of Binarized Deep Neural Networks,” *arXiv*, 2017.

[80] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, “SemFix: Program Repair via Semantic Analysis,” in *ICSE ’13*.

[81] T. Nguyen, T. Antonopoulos, A. Ruef, and M. Hicks, “Counterexample-Guided Approach to Finding Numerical Invariants,” in *FSE’17*.

[82] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest, “DIG: A Dynamic Invariant Generator for Polynomial and Array Invariants.”

[83] S. Padhi and T. Millstein, “Data-Driven Loop Invariant Inference with Automatic Feature Synthesis,” *arXiv*, 2017.

[84] D. Qi, H. Nguyen, and A. Roychoudhury, “Path Exploration using Symbolic Output,” in *TOSEM’13*.

[85] N. Qian, “On the Momentum Term in Gradient Descent Learning Algorithms,” in *Neural Networks’99*.

[86] E. Rodríguez-Carbonell and D. Kapur, “Automatic Generation of Polynomial Invariants of Bounded Degree using Abstract Interpretation,” *Science of Computer Programming*, 2007.

[87] S. Ruder, “An Overview of Gradient Descent Optimization Algorithms,” *arXiv*, 2016.

[88] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning Internal Representations by Error Propagation*, 1985.

[89] P. Rümmer and T. Wahl, “An SMT-LIB Theory of Binary Floating-Point Arithmetic,” in *SMT’10*.

[90] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, “A Symbolic Execution Framework for JavaScript,” in *SP’10*.

[91] P. Saxena, P. Poosankam, S. McCamant, and D. Song, “Loop-Extended Symbolic Execution on Binary Programs,” in *ISSTA’09*.

[92] K. Sen, G. Necula, L. Gong, and W. Choi, “multiSE: Multi-path Symbolic Execution,” in *FSE’15*.

[93] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, “NEUZZ: Efficient Fuzzing with Neural Program Smoothing,” *arXiv*, 2018.

[94] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, “SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis,” in *SP’16*.

[95] S. F. Siegel, M. Zheng, Z. Luo, T. K. Zirkel, A. V. Marianiello, J. G. Edenhofner, M. B. Dwyer, and M. S. Rogers, “CIVL: The Concurrency Intermediate Verification Language,” in *SC’15*.

[96] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting,” *JMLR’14*.

[97] X. Xie, B. Chen, Y. Liu, W. Le, and X. Li, “Proteus: Computing Disjunctive Loop Summary via Path Dependency Analysis,” in *FSE’16*.

[98] Y. Yao, L. Rosasco, and A. Caponnetto, “On Early Stopping in Gradient Descent Learning,” *Constructive Approximation*, 2007.

[99] Y. Yu, H. Qian, and Y.-Q. Hu, “Derivative-Free Optimization via Classification,” in *AAAI’16*.

[100] Y. Zheng, X. Zhang, and V. Ganesh, “Z3-str: A Z3-Based String Solver for Web Application Analysis,” in *FSE’13*.

[101] M. Zitser, R. Lippmann, and T. Leek, “Testing Static Analysis Tools using Exploitable Buffer Overflows from Open Source Code,” in *ACM SIGSOFT Software Engineering Notes*, 2004.