XSTRESSOR: Automatic Generation of Large-Scale Worst-Case Test Inputs by Inferring Path Conditions

Charitha Saumya, Jinkyu Koo, Milind Kulkarni, Saurabh Bagchi
School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN
{cgusthin, kooj, milind, sbagchi}@purdue.edu

***Abstract*—An important part of software testing is generation of worst-case test inputs, which exercise a program under extreme loads. For such a task, symbolic execution is a useful tool with its capability to reason about all possible execution paths of a program, including the one with the worst case behavior. However, symbolic execution suffers from the path explosion problem and frequent calls to a constraint solver, which make it impractical to be used at a large scale. To address the issue, this paper presents XSTRESSOR that is able to generate test inputs that can run specific loops in a program with the worst-case complexity in a large scale. XSTRESSOR synthetically generates the path condition for the large-scale, worst-case execution from a predictive model that is built from a set of small scale tests. XSTRESSOR avoids the scaling problem of prior techniques by limiting full-blown symbolic execution and run-time calls to constraint solver to small scale tests only. We evaluate XSTRESSOR against WISE and SPF-WCA, the most closely related tools to generate worst-case test inputs. Results show that XSTRESSOR can generate the test inputs faster than WISE and SPF-WCA, and also scale to much larger input sizes.**

***Index Terms*—Symbolic execution, Worst-case complexity, Automatic program testing, Stress testing, Program Synthesis**

I. INTRODUCTION

Test input generation plays an important role in software development and maintenance. With the rapid increase in complexity and scale of modern software, test input generation has become far more challenging. Most of existing software test suites have focused on *code coverage*, attempting to exercise as much of a code base as possible [28], [9], [4], [17], [10], [8], [34]. Another challenge in testing, though, is the generation of *stress tests*. Rather than focusing on code coverage, stress tests focus on stressing key portions of the code (*e.g.*, causing a critical loop to execute numerous times), exposing potential performance bottlenecks and bugs that otherwise might only manifest when software is deployed at large scales. Stress testing is also useful for identifying possible security threats such as denial-of-service via algorithmic complexity attacks [11]. There has been comparatively littlex work in generating stress tests [39], [3], [2]. The particular challenge that we address in this paper is generating *large-scale, worst-case inputs*. Here, the scale is defined specifically as the number of elements in a program input, such as the dimension of an input matrix. The worst-case inputs in our context are the ones that execute a specific target basic block a maximal number of times. Generating such specific inputs is indeed challenging because it requires deep understanding of program behaviors.

Symbolic execution is a popular test generation technique that can be used to target particular behaviors in a program [21]. Symbolic execution can reason about all possible execution paths of a program and generate a concrete input that will exercise a certain path using a constraint solver [13], [6]. However, symbolic execution suffers from the *path explosion problem*, which means that the number of paths to explore increases exponentially when the input size grows. For example, when Dijkstra's shortest path algorithm is symbolically executed using KLEE [8], input sizes 3, 4, and 5 (meaning the number of nodes in a graph) result in 4, 56, and 2592 feasible paths, respectively, which require 22, 131, and 5016 number of constraint solver calls, correspondingly. With such rapid increases, finding test inputs at a large scale that lead to worst-case behavior can be impractical with standard symbolic execution methods.

There have been a few attempts in recent years to tackle the path explosion problem and generate large-scale inputs by effectively pruning the search space in symbolic execution [7], [26]. These techniques work by guiding the symbolic execution along a subset of feasible paths according to some criteria. The basic idea, as outlined in WISE [7], is to provide *generators* that choose a specific direction for each branch as symbolic execution proceeds, guiding the search along a particular path. More specifically, WISE is interested in branching policies that pick out worst-case paths, avoiding the path explosion problem by pruning paths that will *not* lead to the worst-case. By way of analogy, think of these generators as branch predictors that predict which direction the branch should go to follow the worst-case path. The predictors are made by fully exploring the space of paths at small scales, where the path explosion problem is manageable, and using this information to predict how branches behave along the worst-case path at large scales.

WISE [7] proposes a few simple generator strategies (such as "always `true`" or "`false` only if `true` is not feasible") for selecting worst-case paths, while later work, such as SPF-WCA [26], proposes more expressive generators based on fixed-length histories of branch decisions that occur at small scales. While these approaches prune the search space (often to a single path that needs to be explored), they still require that the program be symbolically *executed* at the desired large scale, possibly necessitating solver invocations at every branch point. This can easily lead to significant run-time and memory overhead. For example, if *binary tree search* is guided

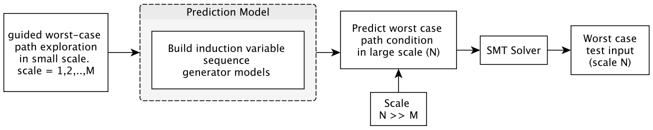

Fig. 1: Overview of XStressor showing its main steps; (i) training phase – which generates induction variable sequence generator models (ii) prediction phase – which generates the worst-case input for the large-scale execution.

along worst-case paths using WISE's branch policy, it still has to make 55 solver calls when searching a tree with just 10 elements, and the number grows quadratically with scale. Hence, even though the path explosion problem is solved, the *scaling* problem remains: making a large number of solver calls, each with an increasing number of constraints, is still impractical.

In this paper, we present **XStressor**, a scalable technique to generate test inputs that will trigger the worst-case performance of programs, in a large-scale execution.

1) Unlike WISE and SPF-WCA, XStressor *does not* require symbolically executing a program at the desired large scale to generate the test input. Instead, XStressor builds a model of program behavior and *synthesizes* a large-scale path condition without executing the program. Hence, to generate a large-scale test input, the constraint solver needs to be invoked *only once*, removing a key scalability bottleneck.
2) Like WISE [7] and SPF-WCA [26], XStressor uses small-scale behavior to predict large-scale worst-case behavior. XStressor is parameterized on the technique used to build its program models. Our implementation makes use of a novel recursive modeling technique to capture *scale-dependent* branching behavior (*e.g.*, a branch that, at scale N takes $(N/2-1)$ `true` branches before taking a `false` branch) that occur within nested loops. Hence, XStressor, instantiated with our recursive modeling technique, is capable of predicting worst-case inputs for a larger class of programs than WISE or SPF-WCA.

XStressor, summarized in Figure 1, works in two key stages: the *model building stage* and the *prediction stage*. First, in the model building stage, XStressor utilizes a series of small-scale runs to observe the scaling behavior of key branching points in a program, and learns a model of this scaling behavior. The model characterizes the behavior of the program in two ways: (i) determining how many times each branch is executed; and (ii) learning the relationship between induction variables and the branch decisions of the program. Second, in the prediction stage, XStressor uses the model to *directly predict the worst-case path condition at a given large scale*. This path condition is solved, using a constraint solver, to generate the worst-case, large-scale test input. Crucially, XStressor *does not need to execute the program at large scale* to generate this path, and hence needs to invoke the solver only once to generate the large-scale path condition.

We evaluate XStressor in two ways: first, like WISE and SPF-WCA, we examine several small programs and show that XStressor can generate worst-case inputs faster than the prior two approaches. Second, we look at two case studies of real-world programs to show that XStressor can generate worst-case inputs for realistic functions.

Contributions

To summarize, the contributions of this paper are:

- A new framework, XStressor, that uses small-scale models of worst-case program behavior not to drive branch decisions at large scales (as WISE and SPF-WCA do) but instead to *directly generate the path condition* that must be solved for large scales. This allows large scale test inputs to be generated very quickly.
- An instantiation of XStressor's modeling component that uses a novel recursive modeling technique to capture complex nested branch patterns that arise from nested control structures.
- An evaluation of XStressor, showing that it can generate large-scale, worst-case inputs faster than prior work on a wide variety of test programs.

II. Background

This section first explains dynamic symbolic execution, then discusses WISE [7], and SPF-WCA [26] which are two symbolic execution based approaches for worst-case input generation.

A. Dynamic symbolic execution

Dynamic symbolic execution [21] is a technique of executing programs using symbolic variables instead of concrete values as input. With some variables declared as symbolic, dynamic symbolic execution can explore all feasible paths in a program, computing a *path condition* for each explored path. A path condition is essentially the conjunction of branching conditions that are true along the path. This path condition can be solved using a SMT solver [6], [13] to find a concrete input which exercises that path. Unfortunately, symbolic execution suffers from the *path explosion problem*, wherein the number of paths grows exponentially with the number of branches in the program [32], [16], [37].

B. WISE and SPF-WCA

WISE [7] tackles the path explosion problem by using heuristics to prune the search space of possible paths. The basic strategy of WISE is as follows. First, WISE uses symbolic execution to explore all paths at small scales (where the search space of paths is tractable), then learns a set of *generators* for each branch in the program. These generators describe which path to take at a branching point such that worst-case path is guaranteed to be explored. WISE proposed several generator strategies, such as always `true` or always `false`, but some more complex strategies (such as alternating between the two) were not supported. SPF-WCA [26] extended WISE with more complex generator strategies by considering the decision history for a particular branch. One class of programs

```
1 void isort(int* arr, int len)
2     int i = 1;
3     while(i < len)
4         int x = arr[i];
5         int j = i−1;
6         while(j >= 0 && arr[j] > arr[i])
7             arr[j+1] = arr[j];
8             j−−;
9         arr[j+1] = x;i++;
```

Fig. 2: Insertion sort pseudocode

that neither WISE nor SPF-WCA can handle are those with *scale-based* branch policies, *i.e.*, branches whose worst-case behavior changes as the input scale increases (such as Boyer-Moore [5] string search algorithm, explained in section VI-D). XSTRESSOR's modeling technique is capable of capturing these types of worst-case branch behaviours. Some of the branch policies proposed by WISE and SPF-WCA require the help of a SMT-solver to decide which path to take at a branching point. This type of branching policy limits the ability of WISE and SPF-WCA to generate large-scale inputs because calling SMT solvers at each step is time consuming. This is visible in the *binary tree search* program as described in section I. XSTRESSOR avoids this problem through a novel *path synthesis* strategy (Sections IV-A and IV-B). It *predicts* what the large scale path will be and invokes the solver only once to generate the necessary input, rather than symbolically executing the program at large scales (Section IV-E).

III. OVERVIEW OF XSTRESSOR

This section describes XSTRESSOR's overall approach to synthesizing worst-case path conditions, using the *insertion sort* program in Figure 2 to explain the key concepts. Note that we use the following definition of "worst-case path":

Worst-case path: Let π be the set of all possible paths in a program for input size N. We define a program path $\pi' \in \pi$ to be a worst case path if it causes a time consuming basic block B inside a target loop L to have the maximum number of executions, among all paths in π. If B is inside a nested loop within L, we take the cumulative execution count. The loop L of interest for our problem of stress testing is generally the most time consuming loop (or "hot loop") in the program.

Consider symbolically executing *insertion sort* by setting a fixed size input array to be symbolic. The worst case occurs when the next element in the input array needs to be swapped to the beginning of already-sorted part of the array. In other words the condition $arr[j] > arr[i]$ is `true` for all executions of the conditional statement at line 6 in Figure 2. This causes the swap operation (basic block from lines 7-8) to execute the maximum number of times for a given input size. XSTRESSOR's modeling technique to infer the worst-case path condition depends on the following key observations.

1) Variables i, j in the condition $arr[j] > arr[i]$ are loop induction variables. In this example, variable j is the inner loop induction variable and i the outer loop induction variable.
2) We observe that in the sequence of branch conditions added by the `true` branch of $arr[j] > arr[i]$ during the worst-case execution, variables i and j follow predictable patterns which can be expressed as sequences. These sequences are dependent on the nesting levels of the two loops and how many times the inner loop is executed for each iteration of outer loop. For input size 4 as an example, the path condition can be expressed as $(arr[0] > arr[1]) \wedge (arr[1] < arr[2]) \wedge (arr[0] > arr[2]) \wedge (arr[2] > arr[3]) \wedge (arr[1] > arr[3]) \wedge (arr[0] > arr[3])$. Note that here all the conditions are placed in their execution order. For these sequence of conditions, the index variable j, corresponding to the inner loop, will take on the values $0, 1, 0, 2, 1, 0$, and variable i will have the values $1, 2, 2, 3, 3, 3$. Both these sequences of numbers are structured and can be produced by a generator function.
3) For a given large scale N, if we have a sequence generator model that can generate the correct i and j values described above we can synthesize the worst-case path constraints for input scale N, *without needing to do symbolic execution at the large scale*.

Note that in this simple example, there is only a single loop that we want to scale up. However, in real-world programs there will be multiple loops. Through out-of-band mechanisms like profiling, it can be determined which are the "hot" loops, *i.e.*, computationally expensive loops, and XSTRESSOR can be applied to generate inputs to scale each of these loops, one at a time.

XSTRESSOR leverages the insights from above in its operation. First, it fully explores the execution space at small scales. It uses this information to build a set of predictive models that can describe the worst-case path conditions for a program. These models are basically a set of sequence generator functions that captures the scaling patterns of loop induction variables and how those variables determine specific values in the path condition clauses inside loops (see Sections IV-A and IV-B). XSTRESSOR then uses these models to directly synthesize the path condition for a large-scale input (Section IV-E) without performing symbolic execution.

IV. DETAILED DESIGN

In this section we describe the algorithms used by XSTRESSOR in detail. First we define the following key terms used in our algorithm description.

Definition 1. *A **symbolic branch** is as an edge $E = \{V_c, V_s\}$ in the control flow graph of the input program where V_c is a conditional statement whose outcome depends on a symbolic variable, and V_s is a statement in the program.*

Definition 2. *A **symbolic assignment** statement is of the form $a := m$ where m contains a symbolic input variable.*

Definition 3. *A **constraint generator point** (CGP) is any symbolic branch or symbolic assignment statement that lies within the target loop in our given program.*

Any conditional statement that uses a symbolic variable has two constraint generator points, one each for `true` and

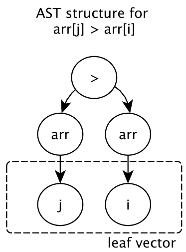

Fig. 3: Leaf vector for CGP $arr[j] > arr[i]$ in insertion-sort

`false` branches. Each constraint generator point is capable of adding a constraint to the path condition. For the condition $arr[j] > arr[i]$ in insertion sort example (Figure 2) the possible constraint generator points correspond to the constraints $arr[j] > arr[i]$ and $\neg(arr[j] > arr[i])$.

Definition 4. *Consider the abstract syntax tree (AST) representation of a constraint generated by a constraint generator point. A* **leaf vector** *is simply the non-constant leaf values of this AST collected in order (from left to right).*

Consider the AST shown in Figure 3. We can generate different concrete conditions by changing the values of i and j in $arr[j] > arr[i]$, *i.e.*, the leaf vector will be [0,1] for $arr[0] > arr[1]$, and it will be [1,0] for $arr[1] > arr[0]$. For our modeling technique to work, each CGP must generate fixed-length leaf vectors (for the insertion sort example, this length is 2). In Section IV-D we describe a simple transformation that makes all CGPs have fixed-length leaf vectors. XSTRESSOR's algorithm consists of the following steps.

1) Exhaustive symbolic execution at small scales is used to generate worst-case path conditions.
2) Using information from step 1, a set of generator functions are synthesized. Given some CGP and some input scale, these functions can be used to predict the sequence of numbers produced by each variable in this CGP's leaf vector. This allows us to predict the sequence of leaf vectors produced by this CGP. These models are called *induction variable sequence generators (ISG)*. For example, for the CGP $arr[j] > arr[i]$ these generator functions can predict the sequences generated by variables j, i for any input scale.
3) For a user-given large scale, for each CGP, we compute the sequence of leaf vectors produced by this CGP using the ISG models. Finally the worst-case path condition is synthesized using the predicted leaf vectors. The path condition can then be solved using a SMT solver to find a concrete worst-case input for the given large-scale.

Section IV-A describes ISGs in more detail, and Sections IV-B and IV-C explain one specific instantiation of ISG modeling that XSTRESSOR supports.

A. Induction variable sequence generators

While on its surface, XSTRESSOR seems to be similar to WISE and SPF-WCA—it uses exhaustive execution at small scale to build models of behavior at large scale—it has a fundamental difference. WISE and SPF-WCA are interested in *guiding symbolic execution* at large scale—their models need only generate branch direction decisions, while the paths are explored directly by symbolic execution. In contrast, XSTRESSOR is interested in *completely avoiding symbolic execution* at large scale. Hence, XSTRESSOR's models must predict not only the direction of branches at large scales but the specific constraints in the path so that the path can be solved to generate a large input. We call these models *induction variable sequence generators*, or ISGs.

In order to generate the constraint clauses added by some CGP for some input scale, we need to model the sequences generated by each variable in this CGP's leaf vector. Recall that a leaf vector is the set of variables in the leaf nodes of the AST for a given condition. We observe that for programs with loops, the variables in the leaf vectors of these CGPs are functions of loop induction variables. If the variation of the loop induction variables at different scales can be modeled, then the variation of leaf vectors of generator points can also be modeled. Recall from Section III that in order to generate the worst-case path condition for *insertion sort*, we need to model the variation of variables j and i which are in the leaf vector for the condition $arr[j] > arr[i]$. We model this using the induction variable sequence generators (ISG).

Consider a loop with k nesting levels, and a CGP p within this loop. Assume that p's leaf vector contain some variable i_p. Now take all the constraint clauses generated by p during some worst-case execution. If we record the value of i_p from all these clauses in the order they are executed, it will generate a sequence of numbers.

For example, consider the worst-case execution for *insertion sort* (Figure 2). The CGP $arr[j] > arr[i]$ at line 6 has the leaf vector $[j, i]$ according to our definition. Now if we sample the value of j for input scale 5, it will generate the sequence $0, 1, 0, 2, 1, 0, 3, 2, 1, 0$. We call this sequence an *induction variable sequence*. A generator function G_{i_p} that can generate this sequence for any input scale is defined as the ISG for variable i_p in the leaf vector of CGP p. Intuitively, the induction variable sequences show a recursive structure for programs with loops, meaning that these sequences can be generated by calling a set of basic sequence generator functions recursively. XSTRESSOR's ISG models are based in this key insight (see Section IV-B).

Sequence Notation : We use parentheses (*"(" and ")"*) to represent the loop nesting level inside a sequence. *"("* and *")"* denote increasing and decreasing the nesting level by one respectively. According to this notation, the sequence in above example can be represented as $((0), (1, 0), (2, 1, 0), (3, 2, 1, 0))$.

B. Constructing ISGs using a context-free grammar

We observe that complex induction variable sequences can be generated by recursively calling simpler sequence generators. XSTRESSOR captures this recursive structure and builds a scale independent representation of the sequence by using a context-free grammar [19]. To understand the semantics of this grammar (shown in Figure 4), consider a non-nested loop

$$P \rightarrow I|C \quad (1)$$

$$I \rightarrow incre(X, X, d) \quad (2)$$

$$C \rightarrow const(X, X) \quad (3)$$

$$X \rightarrow P|x \quad (4)$$

Fig. 4: Context-free grammar used by XSTRESSOR to model induction variable sequence generators

L and CGP p inside this loop. Assume p has variable i_p in its leaf vector. XSTRESSOR's ISGs use the observation that the variable i_p will generate one of the following sequences during the worst-case execution of L,

- Variable i_p can be constant for all iterations of loop L during the worst case. In this case we identify the sequence i_p generates as a **constant** sequence. A constant sequence can be denoted as $C_{k,x}$ where k is the constant value and x is how many times k is repeated in the sequence.
- Variable i_p can start with value i_{start} and increase/decrease by amount d until it reaches the value i_{end}. In this case i_p generates a **increment** sequence. A increment sequence can be denoted as $I_{i_{start},i_{end},d}$ where i_{start}, i_{end} and d are the initial value, the final value and the stride of the sequence.

Note that for above definition, L must be a non-nested loop. If L is nested, these basic sequences get nested on each other. XSTRESSOR uses the context-free grammar shown in Figure 4 to describe this nested structure of sequences. The goal is to use this grammar to describe an ISG for any given induction variable sequence. The grammar is based on 2 main functions, $incre$ and $const$, defined in Algorithms 1 and 2 respectively.

- $incre$ takes in three parameters as arguments, two equal-length sequences (denoted by X_1 and X_2 in the grammar) and some value d. Assume that X_1 and X_2 have l elements each, and the values at jth position in X_1 and X_2 are x_j^1 and x_j^2 respectively. Then $incre$ returns the sequence $I_{x_0^1,x_0^2,d} \oplus I_{x_1^1,x_1^2,d} \oplus \cdots \oplus I_{x_j^1,x_j^2,d} \cdots \oplus I_{x_{l-1}^1,x_{l-1}^2,d}$. Here $\oplus$ denotes concatenation of two sequences.
- $const$ takes in two equal-length sequences as arguments. Let's denote these two sequences as Y_1 and Y_2. Assume that Y_1 and Y_2 have l elements each, and the values at jth position in Y_1 and Y_2 are y_j^1 and y_j^2 respectively. Then const returns the sequence $C_{y_0^1,y_0^2} \oplus C_{y_1^1,y_1^2} \cdots \oplus C_{y_j^1,y_j^2} \cdots \oplus C_{y_{l-1}^1,y_{l-1}^2}$.

Note that argument X in $incre$ and $const$ function can be a single number (*i.e.* grammar rule 4 in figure 4), in which case it can be treated as a sequence of just a single number. Grammar rule 1 says that any induction variable sequence in this model can be described using $incre$ or $const$ functions with some reduced sequences as their arguments. This design allows XSTRESSOR to recursively define complex induction variable sequences. To elaborate this idea consider the following examples.

Example 1: Consider the CGP $arr[j] > arr[i]$ in insertion sort (Figure 2). For worst-case execution at input scale 5, variable i will generate the sequence $((1), (2, 2), (3, 3, 3), (4, 4, 4, 4))$.

Algorithm 1: $incre(X_1, X_2, d)$

```
list L ← empty
i ← 0
for i < length(X_1) do
    j ← X_1[i]
    for j ≤ X_2[i] do
        L.append(j)
        j ← j + d
    end
    i ← i + 1
end
return L
```

Algorithm 2: $const(X_1, X_2)$

```
list L ← empty
i ← 0
for i < length(X_1) do
    j ← 0
    for j < X_2[i] do
        L.append(X_1[i])
        j ← j + 1
    end
    i ← i + 1
end
return L
```

This can be represented as a sequence of calls to $const$ function; $const(1, 1) \oplus const(2, 2) \oplus const(3, 3) \oplus const(4, 4)$. This can be expressed as a single function call to $const$ by expressing the arguments as lists. This gives us the exprssion $const([1, 2, 3, 4], [1, 2, 3, 4])$. Now the argument $[1, 2, 3, 4]$ can be generated using a call to $incre$ as $incre(1, 4, 1)$ because the sequence starts with 1, ends with 4 and has a stride of 1. Therefore the generator simplifies to the expression, $const(incre(1, 4, 1), incre(1, 4, 1))$. If same procedure is repeated for input scale 4, we can get the generator function, $const(incre(1, 3, 1), incre(1, 3, 1))$. Therefore for some general input scale N, ISG for variable i can be expressed as,

$$const(incre(1, N-1, 1), incre(1, N-1, 1))$$

Example 2: Now consider the sequence generated by variable j in CGP $arr[j] > arr[i]$. This is sequence is $((0), (1, 0), (2, 1, 0), (3, 2, 1, 0))$ for worst-case execution at input scale 5. This can be represented as a sequence of call to $incre$ function; $incre(0, 0, -1) \oplus incre(1, 0, -1) \oplus incre(2, 0, -1) \oplus incre(3, 0, -1)$. Here subsequence $[3, 2, 1, 0]$ is simply $incre(3, 0, -1)$ because it has $i_{start} = 3$, $i_{end} = 0$ and $d = -1$. The sequence of calls then can be expressed as $incre([0, 1, 2, 3], [0, 0, 0, 0], -1)$ As in Example 1 this can be further simplifies to the expresion $incre(incre(0, 3, 1), const(0, 4), -1)$. For scale 4, the generator function we get would be $incre(incre(0, 2, 1), const(0, 3), -1)$. And for scale N the generator for variable j can be expressed as,

$$incre(incre(0, N-2, 1), const(0, N-1), -1)$$

Numerical values in these general models are called *parameters* of the ISG. In Example 2 these parameters are $0, N-1, 1, 0, N-1, -1$.

C. Learning an ISG

To build a general ISG, first the structure of the ISG needs to be identified. This is done as illustrated in Examples 1, 2 above. We infer the generator used for a specific sequence by performing a bottom-up parse of the sequence (with brackets denoting loop boundaries). For a sequence of numbers, we summarize based on whether the numbers are constant or incrementing. For a sequence of functions, we summarize based on a single call to that function with sequences representing the various parameters in the sequence of functions. These sequences are then summarized using $const$ and $incre$ functions as necessary. This procedure is called *generator inference.* Generator inference only gives us a generator that works for input scale n. Next, a general ISG must be computed using model fitting. This can be done by inferring the generators for a set of input scales (say $1, 2, \ldots, M$) and fitting some function model for that ISG's parameters. XSTRESSOR's ISG synthesis algorithm works as follows.

Consider a program P and a CGP p inside some target loop of P. Assume that i_p is one of p's leaf vector variables. Assume S is the set of induction variable sequences generated by i_p during the worst-case execution for input scales $1, 2, \ldots, M$. XSTRESSOR uses the parsing steps in described above to compute $G_{i_p}(1), G_{i_p}(2), \ldots, G_{i_p}(M)$. Here $G_{i_p}(k)$ is the ISG computed for input scale k. Then XSTRESSOR uses this set of ISGs to compute a general ISG $G_{i_p}(N)$. This is done using model fitting for ISG parameters as explained in Example 2.

We found that for a large class of programs ISG parameters can be described using a polynomial functions of the input scale.[1] In Example 2, the second ISG parameter has the values $1, 2, 3$ for input scales $3, 4, 5$; therefore the polynomial model is $N-2$ for this parameter. It should be noted that there can be multiple polynomials with different orders that perfectly fit a given set of data points. To reduce model complexity, we select the lowest-order polynomial that perfectly fits *all* the data points in scales $1, 2, \ldots, M$. Therefore we need a threshold of k data points in small scales such that model computed using scales $1, \ldots, k$ is the same as model computed using scales $1, \ldots, k+1$. Procedure described above can be used to build an ISG for a single leaf vector variable (*i.e.* i_p). This can be repeated for all other variables in the leaf vector to build a complete generator model for the given CGP. XSTRESSOR's ISG synthesis procedure is summarized in algorithm 3.

D. Handling symbolic assignment statements

In Section IV-A we mentioned that any variable in a leaf vector can be described recursively using two basic sequences; *increment* and *constant* sequences. However this is not the case when some leaf vector variable is updated inside the loop. This

[1]ISG models can be built using other kinds of functions (e.g., exponential function), given that the function perfectly fits all the data points

```
1 for(int i=0; i<N; i++){
2     if(A[i]>5){
3         A[i+1] = A[i]+1;
4     }
5     else break;
6 }
```

Fig. 5: Example source code with symbolic assignment statements

causes the corresponding CGP to generate constraint clauses with different abstract syntax tree (AST) structures and we can no longer define a fixed length leaf vector for this CGP because each constraint has a different AST structure. To elaborate more on this issue consider the example code in Figure 5. Assume that A is a symbolic array with length 3. Worst-case occurs when condition at line 2 is true for all iterations of the *for* loop. The condition generated by the CGP $A[i] > 5$ is $A[0] > 5$ during the first iteration. However there is a read-after-write dependence between the conditional check at line 2 and the assignment statement at line 3. Therefore in the second iteration the condition will be $A[0] + 1 > 5$. The third iteration will produce the constraint $(A[0]+1)+1 > 5$, which simplifies to $A[0] + 2 > 5$. Notice that this CGP no longer produce a fixed length leaf vector because for $A[0] > 5$ leaf vector is $[0, 5]$ and for $A[0] + 1 > 5$, it is $[0, 1, 5]$. Therefore XSTRESSOR's ISG models no longer model this CGP because leaf vectors do not have a fixed length.

XSTRESSOR solves this problem by converting all the assignment statements in the program to *dynamic single assignment form* [35]. In dynamic single assignment form, each program variable is assigned only once. This is done by introducing a version number for each variable. Every time a static variable is assigned to, we increment its associated version number and this assignment is converted to a unique constraint using this updated version number. When this variable is accessed again, the most recent version is accessed. Intuitively, each variable becomes a vector, and each assignment to that variable is made to a different element of that vector. The version numbers are, essentially, indices of this vector. Using dynamic single assignment form causes each CGP to generate constraint clauses with a fixed AST structure, hence these clauses will have a fixed length leaf vector.

Example: Now assume that in previous example (Figure 5), each variable will have an additional array index (*i.e.* version number), therefore variable $A[i]$ now becomes $A[i][j]$ where i is the actual array index and j is the version number for array element $A[i]$. During the first iteration of the loop constraint produced by the CGP $A[i] > 5$ is $A[0][0] > 5$, because $A[0]$ is accessed and its version is not updated yet. Assignment at line 3 will produce the constraint $A[1][1] \equiv A[0][0] + 1$, because $A[1]$ is assigned and its version number is updated to 1. $A[0]$ is accessed and its most recent version is 0. During the second iteration $A[i] > 5$ will produce the clause $A[1][1] > 5$. Here $A[1]$ is accessed and its most recent version is 1 because $A[1]$ was assigned a new value during iteration 1. Similarly the assignment at line 2 will produce the clause $A[2][1] \equiv$

$A[1][1] + 1$. Note that now each constraint clause generate by CGP $A[i] > 5$ has a fixed length leaf vector. For $A[0][0] > 5$ this is $[0, 0, 5]$ and for $A[1][1] > 5$ the leaf vector is $[1, 1, 5]$. Constraints generated by the assignment statements have the same property.

E. Online operation at large scale

Synthesizing the worst case path condition for any user given large scale (N) can be done as follows. Consider a constraint generator point p with a leaf vector variables $[i_p^1, i_p^2, \ldots, i_p^k]$. Let $G_{i_p^1}, G_{i_p^2}, \cdots G_{i_p^k}$ be the ISG models for these variables. We can predict the sequence generated by leaf vector variable i_p^m at scale N using its ISG model. This will be simply $G_{i_p^m}(N)$. Recall that the ISG model for a variable can predict the induction variable sequence generated by this variable at any input scale. This prediction routine can be repeated for all the variables in the leaf vector p. Once we have all the sequences, we can rewrite the constraints generated by p at scale N. To get the complete path condition, this procedure must be repeated for all CGPs in the program. Finally, the generated path condition, which is a conjunction of all the clauses, is solved using any standard SMT solver—XSTRESSOR uses Z3 [14]. Importantly, XSTRESSOR does not use symbolic execution at large scale N and invokes the SMT solver only once for that scale.

Algorithm 3: $BuildISG(grammar\ C, sequence\ S[])$

```
list G ← empty
for for each scale i in 1, 2, ... M do
    g ← InferGenerator(C, S[i])
    G.append(g)
end
G' ← ModelFitting(G)
return G'
```

V. IMPLEMENTATION

We used KLEE [8] as our symbolic execution platform to perform exhaustive symbolic execution execution in small scale. Then worst-case paths were identified according to the criteria described in section III. We developed a Python tool[2] (2500 LOC) for constructing the ISG models described in Section IV. Our prototype implementation only attempts to learn polynomial models for ISG parameters (our tool will fail if worst-case behavior requires more complex models). We used Z3 [13] as our constraint solver and Z3's Python interface was used for constraint processing work.

A Python implementation of the target program was manually instrumented to collect the conditions generated by constraint generator points (*i.e.*, conditional and assignment statements). Then the benchmarks were run using the small-scale, worst-case inputs (gathered as described above) to obtain the path conditions as Z3 constraint expressions. For a user-given large scale, the corresponding large scale path condition was synthesized in Z3 format using the ISG models and solved using the Z3 SMT solver to generate a concrete solution. Currently our implementation assumes programs that take integers (such as, Dijkstra) or boolean variables (such as, boolean matrix multiplication) as input. But the approach can be easily generalized to other data types as well depending on the solver capabilities. XSTRESSOR is also capable of modeling array theory based constraints (select, store axioms) [15]. This is important when symbolic variables appear as array indices (`A[a] > 0` where `A` is array variable and `a` is symbolic integer).

VI. EVALUATION

This section presents our evaluation of XSTRESSOR. First, we study its performance versus WISE and SPF-WCA when generating test inputs of various sizes for several benchmarks. Then, sections VI-C and VI-D present the results of applying XSTRESSOR to two real world programs.

A. Experimental setup

We evaluate XSTRESSOR by using it to generate worst-case inputs at varying scales for six different benchmark programs. We chose these programs because they cover the 3 broad classes used in evaluating symbolic execution approaches, including WISE, namely, sorting, searching, and graph algorithms.

1) **Insertion sort**: Our running example, shown in figure 2. Increasing scale means sorting a larger array. Worst case behavior arises when sorting a reverse-sorted array.
2) **Sorted list insert**: Inserting an element into an already sorted array. Increasing scale means increasing array size. In the worst case, a new element is added at the end of the array.
3) **Merging sorted arrays**: Merging two sorted arrays (the "merge" step in merge sort). Increasing scale means merging larger arrays. Worst-case behavior happens when merging two arrays such that elements in the final array alternate between the input arrays. Note that the worst-case behavior for this benchmark requires a branching policy that WISE *cannot* handle because there are multiple worst-case paths.
4) **Binary tree search**: Inserting a set of elements into a binary tree and querying it for a value. Increasing scale means searching larger trees. Worst-case behavior happens when generating a completely unbalanced tree and the queried value is located in the leaf of the unbalanced tree.
5) **Dijkstra's algorithm**: Computing single-source shortest path using Dijkstra's algorithm. Increasing scale means processing a larger graph. Worst-case behavior happens with a graph where there exists a node such that the shortest path visits every node in the graph.
6) **Boolean matrix multiplication**: Multiplication of two dimensional boolean matrices. Increasing scale means increasing matrix size. In worst case, each row in matrix 1 needs to be multiplied with every column in matrix 2.
7) **Traveling Salesman**: Computing the shortest route covering a set of cities such that each city is visited exactly once. In worst case, maximum number of paths between the cities needs to be explored to find the best solution.

[2]this tool is available at https://github.com/charitha22/XSTRESSOR

We ran our experiments on a server consisting of 2 8-core Intel Xeon chips running at 2.7 GHz. Each chip has 20 MB of L3 cache. The machine has 192 GB of RAM.

B. Time to generate large-scale inputs

Table I shows the amount of time it takes for XSTRESSOR, WISE, and SPF-WCA to generate inputs of the specified scale for each benchmark program. For WISE and SPF-WCA, time measurements are based on their publicly available source code. For all 3 techniques, total measured time for each scale includes the time spent in small scale symbolic execution, model building (for XSTRESSOR, models described in Section IV; for WISE, branch policy generators; and for SPF-WCA, history-based branch policies), and generating the large scale input using their respective prediction methods. We see, across the board, that XSTRESSOR is substantially faster at generating large scale inputs than prior work: across scales 10, 20 and 30, XSTRESSOR is 1.06x faster and 2.65x faster than WISE and SPF-WCA, respectively, in the worst case, and 390x faster and 16.3x faster, respectively, in the best case. Moreover, for most of the large scales, WISE and SPF-WCA simply run out of time (OOT) or out of memory (OOM).

XSTRESSOR's advantage is fundamentally attributable to its chief novelty: generating the large-scale input *does not* require performing symbolic execution at large scales. In contrast, WISE and SPF-WCA, even though they aggressively prune the search space, must still perform symbolic execution to generate large-scale inputs, which, in some cases, requires an invocation of the solver at some branches. Requiring symbolic execution at each branch imposes a serious scalability limit on WISE and SPF-WCA, causing timeouts for many benchmarks. In contrast, XSTRESSOR is able to generate inputs up to scale 500 for five of our seven benchmarks. It runs out of time for the largest scales of matrix multiplication due to the limitations of the constraint solver (the number of path constraints at a scale of 200 runs into 8M, cube of the input size). For Traveling Salesmen, XSTRESSOR was unable to compute its ISG models using the small scale executions because the induction variable patterns can not be captured using the context-free grammar described in section IV-B. WISE and SPF-WCA also fails to find a suitable branch policy for this benchmark and ends up exploring all feasible paths.

An interesting case is the *merging sorted arrays* benchmark. Here, WISE times out even for the smallest scale (N = 20). This is because the branch policy required for the worst-case input cannot be captured by WISE's generator templates. As a result, WISE defaults to exploring every possible path, and even at scale 20, the path explosion problem cripples symbolic execution. Note that SPF-WCA does not have this problem due to its history-based branch policies. XSTRESSOR's modeling technique also captures this branch pattern and does not need to explore *any* paths even at scale 10 (modeling can be completed with much smaller scales), avoiding the path explosion problem entirely, and generating worst-case inputs in less than 1 second for even the largest scales.

Breakdown of XSTRESSOR *time*: Table II shows the amount of time XSTRESSOR spends in building ISG models, path synthesis (large scale) and constraint solving stages. Model building time include the time spent in small scale symbolic execution and XSTRESSOR's model building phases; note that this time is the same independent of the final scale of input generation. Path prediction time increases with scale because the number of constraints that have to be predicted is a function of input scale. When the size of the path condition increases, the time taken by the solver also increases. We observed that XSTRESSOR consumes more time in the model building phase compared to WISE and SPF-WCA because XSTRESSOR has to build ISG models for symbolic assignment statements (Section IV-D), which is not a required for WISE and SPF-WCA. However model building is done only once and therefore does not affect XSTRESSOR's performance at large scale. Also, path prediction time and solver time increase when the program has deeply nested loops (as in *Boolean matrix multiplication*), because the number of constraints that needs to be predicted grows rapidly with scale.

C. Case Study 1 : GNU Diffutils cmp

GNU Diffutils is a software package consisting of several programs that can be used to finding differences in files. In this case study we applied XSTRESSOR to the *cmp* utility in *GNU diffutils-3.3*. *cmp* is used to show character offsets/line numbers where two input files differ. Using profiling, we found that most of the work is done in a single function called *block_compare_and_count*. We limited our analysis to this function, as exhaustive symbolic execution of the full *cmp* program is impractical.

Input to the function is two arrays of words. The function has two main loops. In the first loop it compares the two arrays word by word and finds out the first differing word. Then in a second loop the differing two words are compared byte by byte to detect the exact position of the difference. Therefore in the worst case the two arrays of words must match in all byte positions except for the last byte. This input configuration causes both loops to run the maximum number of iterations. We performed exhaustive symbolic execution for input array sizes $2, 3, 4, 5, 6$ and used the worst-case path conditions to build the XSTRESSOR ISG models. We also applied WISE and SPF-WCA to the same program. The results are summarized in Table III. We observed that WISE is not able to prune the entire search tree for this program and hence takes more time at large scale than XSTRESSOR (41 seconds vs. 27 seconds). While SPF-WCA terminated quickly for this program, it failed to produce worst-case inputs, producing suboptimal inputs for all the history sizes (0, 4, 5, 8) we tested on. Note that SPF-WCA provides no guidance on what history size for the previous branches should be selected for worst-case branch behavior.

D. Case Study 2 : GNU grep

GNU Grep is a tool for searching one or more files for lines containing a specified pattern. We used *GNU grep-2.6.1* for our experiments. For this program about 98% of the execution time is spent on a function called *bmexec* that implements

TABLE I: Elapsed time (in seconds) to generate input of scale N for XSTRESSOR, WISE and SPF-WCA. OOT = out of time (12 hour time limit); OOM = out of memory (192GB RAM). X = XSTRESSOR, W = WISE, S - SPF-WCA

Benchmark program	Input scale														
	10			20			30			40			50		
	X	W	S	X	W	S	X	W	S	X	W	S	X	W	S
Insertion sort	1.30	2.14	2.05	1.35	4.12	3.10	1.48	9.83	8.52	1.63	21.03	25.96	1.85	49.57	72.86
Sorted list (insert)	1.63	2.77	2.02	1.64	4.76	3.70	1.64	11.12	11.86	1.64	26.74	31.79	1.65	62.39	86.88
Merging sorted arrays	2.12	826.84	30.82	2.13	OOT	29.88	2.14	OOT	30.1	2.15	OOT	29.9	2.16	OOT	29.96
Binary tree (search)	3.28	4.18	2.41	3.32	6.84	5.31	3.44	16.16	19.38	3.66	33.00	65.05	3.97	56.29	237.67
Dijkstra's	2.49	2.39	2.45	2.86	2.94	11.23	3.60	4.31	158.28	4.84	5.85	602.75	6.32	9.68	1714.26
Boolean matrix multiplication	3.76	6.11	20.35	9.01	42.85	1548.19	23.27	139.34	11649.63	52.86	362.01	OOT	107.87	960.03	OOT
Traveling Salesman	OOT	OOM	OOT	OOT	OOM	OOT	OOT	OOM	OOT	OOT	OOM	OOT	OOT	OOM	OOT
Benchmark program	**Input scale**														
	100			200			300			400			500		
	X	W	S	X	W	S	X	W	S	X	W	S	X	W	S
Insertion sort	3.92	498.92	OOM	14.30	12512.28	OOM	27.61	OOT	OOM	57.03	OOT	OOM	96.73	OOT	OOM
Sorted list (insert)	1.68	553.89	OOM	1.69	12879.05	OOM	1.74	OOT	OOM	1.76	OOT	OOM	1.79	OOT	OOM
Merging sorted arrays	2.20	OOT	30.42	2.30	OOT	32.25	2.40	OOT	35.41	2.49	OOT	43.18	2.57	OOT	46.62
Binary tree (search)	6.05	1043.69	OOM	16.57	16042.95	OOM	29.92	OOT	OOM	58.64	OOT	OOM	103.74	OOT	OOM
Dijkstra's	26.36	27.69	OOT	242.74	193.13	OOT	1438.06	654.39	OOT	4603.48	1581.12	OOT	12830	3099.41	OOT
Boolean matrix multiplication	1279.56	OOM	OOT	22022	OOM	OOT	OOT	OOM	OOT	OOT	OOM	OOT	OOT	OOM	OOT
Traveling Salesman	OOT	OOM	OOT	OOT	OOM	OOT	OOT	OOM	OOT	OOT	OOM	OOT	OOT	OOM	OOT

TABLE II: Time consumption of XSTRESSOR (in seconds).

Program	Time statistic	Scales		
		40	50	100
Insertion sort	Model building	9.28	9.28	9.28
	Path prediction	0.52	0.66	2.01
	Solver	0.10	0.15	0.84
Sorted list Insert	Model building	4.67	4.67	4.67
	Path prediction	0.01	0.02	0.03
	Solver	0.01	0.01	0.02
Merging sorted arrays	Model building	2.08	2.08	2.08
	Path prediction	0.04	0.06	0.13
	Solver	0.03	0.03	0.06
Binary tree search	Model building	10.98	10.98	10.98
	Path prediction	0.33	0.49	1.90
	Solver	0.10	0.16	0.88
Dijkstra's	Model building	25.53	25.53	25.53
	Path prediction	1.92	3.13	17.52
	Solver	0.26	0.41	1.95
Boolean matrix multiplication	Model building	13.68	13.68	13.68
	Path prediction	45.57	92.92	1102.13
	Solver	5.96	14.41	69.32

the *Boyer-Moore* fast string search algorithm [5]. For the subsequent discussion, consider we are searching for a small *pattern* string in a large *text* string.

Boyer-Moore pre-processes the pattern string to build two tables (*delta1* and *delta2*) that helps it to perform maximum possible strides along the text looking for a possible match. *Boyer-Moore* search starts by aligning the pattern string to the beginning of the text and comparisons are made starting from the last character of pattern. If a mismatch is found, the pattern is shifted along the text, where the amount shifted is determined by *delta1* and *delta2* tables. In our experiments we set the text to be a symbolic string and pattern to be the fixed string *abc*. Using small scale symbolic execution we found that in the worst-case, pattern string *abc* is shifted by only one character until the end of the text is reached. XSTRESSOR was able to capture this behavior successfully in its models and we were able to generate large scale text strings that triggers the worst-case behavior. WISE and SPF-WCA did not finish within the given time limit (12 hours) because they were unable to find a simple branch policy that will prune the search space significantly. This is due to the scale-dependent branch behavior of this program, meaning that a certain branch statement in this program take n number of false branches before taking 3 true branches *and* n is scale-dependent.

VII. LIMITATIONS OF XSTRESSOR

There are several limitations of XSTRESSOR that restrict its generality that we enumerate here.

First, the design of XSTRESSOR is geared toward triggering worst-case execution for a target loop. However, the worst-case execution of a complex program, one that has multiple loops, may be some combination of execution behaviors of the different constituent loops, which XSTRESSOR cannot handle. *Second*, XSTRESSOR currently only works for iterative programs; recursive programs that cannot be translated into loops (*e.g.*, tail-recursive functions) cannot currently be modeled. Incorporating recursion will likely require an approach to predicting the structure of recursive calls. *Third*, XSTRESSOR's constraint synthesis step relies on the fact that sequences generated by the leaf vector variables of some CGP can be recursively modeled using a set of fundamental sequence generator functions. *Fourth*, XSTRESSOR is currently geared towards generating the path constraint for a single function or loop nest. We do no special handling for parts of the path that *lead up* to the target loop, nor perform any optimizations to reach the loop efficiently; reaching particular loops in a large code base is out of XSTRESSOR's scope and we can use any of several prior solutions, *e.g.*, [33]. *Finally*, XSTRESSOR, like WISE and SPF-WCA, relies on the model of program behavior constructed at small scales to match the program behavior at large scales. In other words, all these systems rely on some notion of *continuous* behavior. If the large scale behavior of a program is not similar to its small-scale behavior, none of these predictive approaches will work. Note that most of these limitations, other than 2 and 3 (which are specific to XSTRESSOR's path condition synthesis step), are shared with WISE and SPF-WCA.

VIII. RELATED WORK

Analyzing the worst-case behavior or the worst case execution time (WCET) has been been a subject of extensive

TABLE III: Evaluation on the case studies. W = WISE, I = SPF-WCA, X = XSTRESSOR

Application	Model building time (seconds)			Prediction time(seconds)								
				50			100			500		
	W	I	X	W	I	X	W	I	X	W	I	X
GNU cmp	2.98	1.72	4.234	1.40	1.81	1.86	3.31	1.75	4.07	41.24	2.49	26.77
GNU grep	16.99	OOT	22.70	OOT	OOT	96.54	OOT	OOT	674.27	OOT	OOT	29825.23

research [30], [22], [18], [1], [20]. However, automatically generating worst-case test inputs for complex programs is a challenging problem. Symbolic execution has been widely used to analyze all possible program behaviors under different inputs. But in the presence of loops and with increasing input size, using symbolic execution to analyze program behaviors becomes impractical. Concolic execution [24], [32], which is a mix between concrete execution and symbolic execution can avoid prohibitive computational cost associated with full-blown symbolic execution at the cost of reduced coverage. Tools such as SAGE [17], Bouncer [10], KLEE [8], and Pex [34] handled loops by bounding the number of iterations but unable to reason about program behaviours beyond the loop bound.

Guiding path exploration is a natural approach to reducing execution overhead. Both WISE [7] and SPF-WCA [26] uses small scale exhaustive symbolic execution to learn branch policies that will guide symbolic execution only along the worst case paths during the large scale exploration. More recently, PySE [23] was able to capture irregular branching patterns with the help of reinforcement learning. Zhang *et al.* [40] proposed incremental symbolic execution that iteratively deepens exhaustive search depth in a branch tree, pruning out most of similar paths. Even though this approach can be used for programs with much complex worst-case branch patterns, it still needs to invoke a constraint solver at every search step, and there is no guarantee that it can find the worst case complexity. In contrast to these schemes, XSTRESSOR does not require symbolic execution and invokes the constraint solver only once for large scale test generation.

Loop summarization is a recent approach in multi-path loop analysis where the effect of a loop is summarized and reflected as a set of symbolic values [38]. LESE [31] introduces a new symbolic variable, called a trip count, to represent the number of times a loop executes, and then looks for relationship between the trip count and other variables in the program. That way, LESE can model the effects of loops in a program and relate them with features of program input. These summarization techniques are applied at a specific input scale and it is not immediately clear how they can be used for large scale test generation. Further, they do not support handling nested loops, which are common in practice.

Fuzzing techniques have recently been adapted to the problem of generating worst-case program behavior. PerfFuzz [25] and SlowFuzz [29] use heuristics to drive input fuzzers to find long-running execution paths. Fuzzing to generate vulnerabilities or worst-case execution behavior can be more directed when domain knowledge is available, such as through a DSL [27], as shown by [12]. Unlike XSTRESSOR (or WISE and SPF-WCA), they cannot provide any guarantees about finding worst-case behavior, as they are best-effort fuzzers.

Recently, Singularity [36] proposed *pattern* fuzzing for automatically generating worst-case inputs. This idea is based on the fact that worst-case inputs will have some pattern (*e.g.*, the input array is reverse sorted). Singularity is capable of automatically synthesizing an input generator which captures this pattern. There are upsides and downsides to this approach. Singularity does not care about how complex a program is if the worst-case input has a discoverable pattern; in contrast, approaches like XSTRESSOR, WISE, and SPF-WCA may not be able to model such a complex program. On the flip side, XSTRESSOR, WISE, and SPF-WCA are not tied to the particular form of a program's input: if they can model the program behavior, they will generate an input. In contrast, Singularity operates in *input space*: it needs to directly generate inputs, and hence must be instantiated with a library of input generators and knowledge of the domain. So when presented with a program with an unknown, or complex, input domain, white-box approaches may succeed where Singularity fails.

IX. CONCLUSION

Generating large-scale stress test inputs is critical for understanding how programs will behave under load. To ensure that stress tests target critical portions of a program, some form of white-box testing is required. Unfortunately, the most common approach to such test generation, dynamic symbolic analysis, is ill-suited due to its path explosion problem and increased constraint solving overheads. Here we presented XSTRESSOR that solves this problem by using a novel modeling technique which can directly predict the large scale stress test by analyzing the program behavior at small input scales. In this way, large scale inputs can be generated to target specific regions of code without ever performing symbolic execution at large scales. We evaluate XSTRESSOR on a number of benchmarks and show that it is not only faster at generating inputs than the best comparable techniques WISE and SPF-WCA, but can also scale up to much larger sizes of inputs.

X. ACKNOWLEDGMENTS

The authors would like to thank anonymous reviewers for their helpful comments and feedback. This work was partly supported by DOE Early Career Award DE-SC0010295,National Science Foundation under grant numbers CCF-115013 (CAREER), CCF-1439126, CNS-1527262, CNS-1513197 and an unrestricted gift from Adobe Research. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

REFERENCES

[1] H. J. Bang, T. H. Kim, and S. D. Cha. An iterative refinement framework for tighter worst-case execution time calculation. In *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)*, pages 365–372, May 2007.

[2] C. Barna, M. Litoiu, and H. Ghanbari. Autonomic load-testing framework. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, pages 91–100, New York, NY, USA, 2011. ACM.

[3] M. Bayan and J. a. W. Cangussu. Automatic feedback, control-based, stress and load testing. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, pages 661–666, New York, NY, USA, 2008. ACM.

[4] P. Boonstoppel, C. Cadar, and D. Engler. Rwset: Attacking path explosion in constraint-based test generation. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 351–366, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, Oct. 1977.

[6] R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich. The opensmt solver. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'10, pages 150–153, Berlin, Heidelberg, 2010. Springer-Verlag.

[7] J. Burnim, S. Juvekar, and K. Sen. Wise: Automated test generation for worst-case complexity. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 463–473, Washington, DC, USA, 2009. IEEE Computer Society.

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

[9] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM.

[10] M. Costa. Bouncer: Securing software by blocking bad input. In *Proceedings of the 2Nd Workshop on Recent Advances on Intrusiton-tolerant Systems*, WRAITS '08, New York, NY, USA, 2008. ACM.

[11] S. A. Crosby and D. S. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 3–3, Berkeley, CA, USA, 2003. USENIX Association.

[12] M. de Jonge and E. Visser. Automated evaluation of syntax error recovery. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 322–325, Sep. 2012.

[13] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[15] L. de Moura and N. Bjrner. Generalized, efficient array decision procedures. In *2009 Formal Methods in Computer-Aided Design*, pages 45–52, Nov 2009.

[16] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[17] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*, 2008.

[18] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 57–66, Dec 2006.

[19] J. E. Hopcroft, R. Motwani, Rotwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computability*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2000.

[20] D. Kebbal and P. Sainrat. Combining Symbolic Execution and Path Enumeration in Worst-Case Execution Time Analysis. In F. Mueller, editor, *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*, volume 4. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2006.

[21] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'03, pages 553–568, Berlin, Heidelberg, 2003. Springer-Verlag.

[22] J. Knoop, L. Kovács, and J. Zwirchmayr. *Symbolic Loop Bound Computation for WCET Analysis*, pages 227–242. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[23] J. Koo, C. Saumya, M. Kulkarni, and S. Bagchi. Worst-case test generation by reinforcement learning. In *12th IEEE International Conference on Software Testing, Verification, and Validation (ICST)*, April 2019.

[24] E. Larson and T. M. Austin. High coverage detection of input-related security faults. In *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*, 2003.

[25] C. Lemieux, R. Padhye, K. Sen, and D. Song. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 254–265, New York, NY, USA, 2018. ACM.

[26] K. Luckow, R. Kersten, and C. Psreanu. Symbolic complexity analysis using context-preserving histories. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 58–68, March 2017.

[27] K. Mahadik, C. Wright, J. Zhang, M. Kulkarni, S. Bagchi, and S. Chaterji. Sarvavid: A domain specific language for developing scalable computational genomics applications. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pages 34:1–34:12, New York, NY, USA, 2016. ACM.

[28] P. D. Marinescu and C. Cadar. Katch: High-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 235–245, New York, NY, USA, 2013. ACM.

[29] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2155–2168, New York, NY, USA, 2017. ACM.

[30] A. Prantl, J. Knoop, M. Schordan, and M. Triska. Constraint solving for high-level wcet analysis. In *18th Workshop on Logic-based Methods in Programming Environments (WLPE2008)*, 2008.

[31] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 225–236, New York, NY, USA, 2009. ACM.

[32] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[33] J. Strejček and M. Trtík. Abstracting path conditions. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 155–165, New York, NY, USA, 2012. ACM.

[34] N. Tillmann and J. De Halleux. Pex: White box test generation for .net. In *Proceedings of the 2Nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

[35] P. Vanbroekhoven, G. Janssens, M. Bruynooghe, and F. Catthoor. A practical dynamic single assignment transformation. *ACM Trans. Des. Autom. Electron. Syst.*, 12(4), Sept. 2007.

[36] J. Wei, J. Chen, Y. Feng, K. Ferles, and I. Dillig. Singularity: Pattern fuzzing for worst case complexity. In *26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 18)*, November 2018.

[37] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pages 359–368. IEEE, 2009.

[38] X. Xie, B. Chen, Y. Liu, W. Le, and X. Li. Proteus: Computing disjunctive loop summary via path dependency analysis. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 61–72, New York, NY, USA, 2016. ACM.

[39] C.-S. D. Yang and L. L. Pollock. Towards a structural load testing tool. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '96, pages 201–208, New York, NY, USA, 1996. ACM.

[40] P. Zhang, S. Elbaum, and M. B. Dwyer. Automatic generation of load tests. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 43–52, Washington, DC, USA, 2011. IEEE Computer Society.