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

*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. 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 is an important part 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, ensuring to exercise as much portion of a code base as possible [25], [9], [4], [16], [10], [8], [31]. 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 less work in generating stress tests [36], [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 as many as possible. 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 [20]. 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 [12], [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], [24]. 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 [24], proposes more expressive generators based on fixed-length histories 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 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
In this paper, we present TRESSOR, a scalable technique to generate test inputs that will trigger the worst-case performance of programs, in a large-scale execution. TRESSOR makes two key contributions.

1) Unlike WISE and SPF-WCA, TRESSOR does not require symbolically executing a program at the desired large scale to generate the test input. Instead, TRESSOR uses 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 [24], TRESSOR uses small-scale behavior to predict large-scale worst-case behavior. TRESSOR 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, TRESSOR, instantiated with our recursive modeling technique, is capable of predicting worst-case inputs for a larger class of programs than WISE or SPF-WCA.

TRESSOR, summarized in Figure 1, works in two key stages: model building stage and prediction stage. First, in the model building stage, TRESSOR 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, TRESSOR 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, TRESSOR 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 TRESSOR in two ways: first, like WISE and SPF-WCA, we examine several small programs and show that TRESSOR 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 TRESSOR can generate worst-case inputs for realistic functions.

**Contributions**

To summarize, the contributions of this paper are:

- A new framework, TRESSOR, 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 TRESSOR’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 TRESSOR, 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 [24] which are two symbolic execution based approaches for worst-case input generation.

### A. Dynamic symbolic execution

Dynamic symbolic execution [20] is a technique of executing programs using symbolic variables instead of concrete values as input. With some variables declared as symbolic, it can explore all feasible paths in a program computing a path condition for each explored path. 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], [12] to find a concrete input which exercises that path. However symbolic execution suffers from the path explosion problem, wherein the number of paths grows exponentially with the number of branches in the program [29], [15], [34].

### B. WISE and SPF-WCA

WISE [7] tackles the path explosion problem by limiting the exploration at large scale to worst-case path. 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 [24] extended WISE with more complex generator strategies by considering the decision history for a particular branch. One class of programs 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).
2) We observe that in the sequence of branch conditions added
1) Variables condition depends on the following key observations.

This causes the swap operation (basic block from lines 7-8) to
execute the maximum number of times for a given input size.
In other words the condition
occurs when the next element in the input array needs to be
fixed size input array to be symbolic, where the worst case
L loop of executions, among all paths in
π to be a worst case path if it causes a time consuming basic
program for input size
L
strategy (sections IV-A, 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 paths”:

**Worst-case paths:** 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 \(\pi\). 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, where 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 upon the nested 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 \((arr[0] > arr[1]) \land (arr[1] <
arr[2]) \land (arr[0] > arr[2]) \land (arr[2] > arr[3]) \land (arr[1] >
arr[3]) \land (arr[0] > arr[3])\). 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, 2, 1, 0, and
variable \(i\) will have the values 1, 2, 3, 3, 3, 3. Both these
sequences of numbers are structured and can be generated
by some 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 op-
eration. 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 Sec-
tions 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. This
condition is then solved using a constraint solver to generate
an input that will exercise the worst-case path for the given
large scale.

IV. DETAILED DESIGN

In this section we describe the algorithms used by XSTRES-
SOR 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
false branches. Each constraint generator point is capable

```c
void isort(int* arr, int len)
{
    int i = 1;
    while (i < len)
    {
        int x = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > arr[i])
        {
            arr[j+1] = arr[j];
            j--;
        }
        arr[i] = x; i++;
    }
}
```

**Fig. 2: Insertion sort pseudocode**

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 requires
the help of a SMT-solver to decide which path to take at a
branching point. This type of branching policies limits the
ability of WISE, 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, 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).
Definition 4. Consider the abstract syntax tree (AST) representation of a constraint generated by a constraint generator corresponding to the constraints $arr[j] > arr[i]$ and $\neg(arr[j] > arr[i])$.

Fig. 3: Leaf vector for CGP $arr[j] > arr[i]$ in insertion-sort 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 are corresponding to the constraints $arr[j] > arr[i]$ and $\neg(arr[j] > arr[i])$.

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 insertion sort example this length is 2. In Section IV-D we describe a simple transformation which makes all the CGPs to have fixed-length leaf vectors. XSTRESSOR’s algorithm consists of 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, i.e. 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. Path condition can then be solved using a SMT solver to find a concrete worst-case input for the given large-scale.

Note that XSTRESSOR is agnostic to how the models described in step 2 are learned and built, meaning that one can use any fitting model given the fact that they can correctly model leaf vectors for all the CGPs of the program. 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 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 clause 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 $G_{i_p}$ which 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 generators functions recursively. XSTRESSOR’s ISG models are based in this key insight (see Section IV-B).

Sequence Notation: We use square brackets (“[“ 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 induction variable sequence in insertion sort with input size 5 can be represented as ([(0), (1, 0), (2, 1, 0), (3, 2, 1, 0)]). The notation makes it easier to see that the sequence is a combination of simpler sequences.

B. Constructing ISGs using a context-free grammar

XSTRESSOR is agnostic to how its induction variable sequences are built. However, we propose a specific approach
Fig 4: Context-free grammar used by XSTRESSOR to model Induction variable sequence generators

<table>
<thead>
<tr>
<th>Rule</th>
<th>Context-Free Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( P \rightarrow I</td>
</tr>
<tr>
<td>2</td>
<td>( I \rightarrow \text{inc}(X,X,d) )</td>
</tr>
<tr>
<td>3</td>
<td>( C \rightarrow \text{const}(X,X) )</td>
</tr>
<tr>
<td>4</td>
<td>( X \rightarrow P</td>
</tr>
</tbody>
</table>

The grammar is used to build ISGs based on nested sequences. We capture this nested structure and build a scale independent representation of the sequence by reasoning about the structure of the sequence according to a context-free grammar [18]. XSTRESSOR’s induction variable sequence generators are built based on the context-free grammar shown in Figure 4.

To understand the semantics of this grammar, consider a non-nested loop \( L \) 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 \( i_{start} \) and \( i_{end} \) 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, \text{inc}re and const.

- \text{inc}re function (Algorithm 1) 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 \( j \)th position in \( X_1 \) and \( X_2 \) are \( x_{j1} \) and \( x_{j2} \) respectively. Then \text{inc}re returns the sequence \( I_{x_{j1},x_{j2},d} \oplus I_{x_{j1},x_{j2},d} \oplus \cdots \oplus I_{x_{j1},x_{j2},d} \oplus \cdots \oplus I_{x_{l-1},x_{l-1},d} \oplus a \).

- const function (Algorithm 2) 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 \( j \)th position in \( Y_1 \) and \( Y_2 \) are \( y_{j1} \) and \( y_{j2} \) respectively. Then const returns the sequence \( C_{y_{j1},y_{j2}} \oplus C_{y_{j1},y_{j2}} \oplus \cdots \oplus C_{y_{j1},y_{j2}} \oplus \cdots \oplus C_{y_{l-1},y_{l-1}} \).

Both \text{inc}re and const takes sequences (i.e. lists of numbers) as their arguments. These arguments can also be described using the context-free grammar (using rule 2 and 3 in figure 4). Note that argument \( X \) in \text{inc}re and const function can be a single number also (i.e. grammar rule 4 in figure 4) in that 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 \text{inc}re or const functions with some reduced values 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 \((1), (2, 2), (3, 3, 3), (4, 4, 4, 4))\). This can be represented as a sequence of calls to const function; const(\((1, 1) \oplus \text{const}(2, 2) \oplus \text{const}(3, 3) \oplus \text{const}(4, 4)) \). This can be expressed as a single function call to \text{inc}re by expressing the arguments as lists. This gives us the expression
\[
\text{const}(\text{inc}re(1,4,1),1,4,1)
\]
This can be recursively expanded as
\[
\text{const}(\text{inc}re(1,3,1),1,3,1)
\]
for some general input scale \( N \), ISG for variable \( i \) can be expressed as,
\[
\text{const}(\text{inc}re(1,N-1,1),1,N-1,1)
\]

Example 2: Now consider the sequence generated by variable \( j \) in CGP \( arr[j] > arr[i] \). This is sequence \((0), (1,0), (2,1,0), (3,2,1,0))\) for worst-case execution at input scale 5. This can be represented as a sequence of call to \text{inc}re function; \text{inc}re(0,0,−1) \oplus \text{inc}re(1,0,−1) \oplus \text{inc}re(2,1,−1) \oplus \text{inc}re(3,2,−1) \oplus
These sequences are then summarized using denoting loop boundaries. For a sequence of numbers, we can be expressed as \( \text{inc}\left(0, 1, 2, 3\right) \). As in Example 1 this can be further simplifies to the expression \( \text{inc}\left(0, 3, 1\right) \). For scale 4, the generator function we get would be \( \text{inc}\left(0, 2, 1\right) \). And for scale \( N \) the generator for variable \( j \) can be expressed as,
\[
\text{inc}\left(0, N - 2, 1\right, \text{const}\left(0, N - 1\right), -1)
\]

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 Example 2 above. Inference procedure 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 \( \text{const} \) and \( \text{inc} \) 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 sequences for a set of input scales (say \( 1, 2, \cdots, M \)) 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 \( 1, 2, \cdots, M \). XSTRESSOR uses the parsing steps in described above to compute \( \text{G}_{s_p}(N) \). Here \( \text{G}_{s_p}(k) \) is the ISG computed for input scale \( k \). Then XSTRESSOR uses this set of ISGs to compute a general ISG \( \text{G}_{s_p}(N) \). 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. 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 \( 1, 2, \cdots, M \). Therefore we need a threshold of \( k \) data points in small scales such that model computed

\[ \text{inc}(2, 0, -1) \oplus \text{inc}(3, 0, -1) \]. Here subsequence \( [3, 2, 1, 0] \) is simply \( \text{inc}(3, 0, -1) \) because it has \( i_{\text{start}} = 3 \), \( i_{\text{end}} = 0 \) and \( d = -1 \). The sequence of calls then

using scales \( 1, \cdots, k \) is the same as model computed using scales \( 1, \cdots, k + 1 \). 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; \( \text{increment} \) and \( \text{constant} \) sequences. However this is not the case when some leaf vector variable is updated inside the loop. This causes the corresponding CGP to generate constraint clauses with different abstract syntax tree (AST) structures and we cannot 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 conditional check at line 2 and assignment statement at line 3. Therefore in the second iteration the condition will be \( A[0]+1 > 5 \). 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 by converting all the assignment statements in the program to dynamic single assignment form[32]. 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 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:** Returning to the previous example (Figure 5), assume dynamic single assignment form is used now. Note that
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 $A[1][1] ≡ A[0][0] + 1$, 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 $A[2][1] ≡ A[1][1] + 1$. Note that now each constraint clause generated 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. This way XSTRESSOR is able to model the clauses added by CGPs using its ISG models.

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 $[i_1, i_2, \ldots, i_k]$. Let $G_{i_1}, G_{i_2}, \ldots G_{i_k}$ 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 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—we use Z3 in XSTRESSOR [13]. Importantly, XSTRESSOR does not have to use symbolic execution at all at the large scale $N$, and only has to invoke the SMT solver once at this scale.

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

```
list $G$ ← empty
for each scale $i$ in 1, 2, \ldots, $M$ 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 (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 [12] as our constraint solver and Z3’s Python interface was used for constraint processing work.

Python implementations of all the benchmarks were first 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) [14]. This is important when symbolic variables appear as array indices ($A[a] > 0$ where $A$ is array variable and $a$ is a 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. Worst case behavior happens when the new element is to be inserted 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 to 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 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.

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 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 six 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).

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 is the same independent of scale. 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 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 predicted grows rapidly with scale.

C. Case Study 1 : GNU Diffutils

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 techniques we found out 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 (word is 8 bytes long in 64-bit machine). The function has two main loops and 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 the 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 prediction models. Then we used XSTRESSOR models to generate large scale worst-case inputs. 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 takes more time in large scale than XSTRESSOR. SPF-WCA terminated quickly for this program but produced suboptimal worst-case 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
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

<table>
<thead>
<tr>
<th>Benchmarks program</th>
<th>10</th>
<th>20</th>
<th>50</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>W</td>
<td>S</td>
<td>X</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>1.30</td>
<td>2.34</td>
<td>2.08</td>
<td>3.13</td>
</tr>
<tr>
<td>Sorted list (insert)</td>
<td>1.93</td>
<td>4.27</td>
<td>7.05</td>
<td>1.06</td>
</tr>
<tr>
<td>Merging sorted arrays</td>
<td>2.12</td>
<td>826.84</td>
<td>10.82</td>
<td>2.13</td>
</tr>
<tr>
<td>Binary tree (searches)</td>
<td>1.24</td>
<td>4.38</td>
<td>4.41</td>
<td>1.35</td>
</tr>
<tr>
<td>Dijkstra’s</td>
<td>2.49</td>
<td>2.39</td>
<td>2.45</td>
<td>2.86</td>
</tr>
<tr>
<td>Boolean matrix multiplication</td>
<td>3.76</td>
<td>61.11</td>
<td>20.38</td>
<td>9.01</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmarks program</th>
<th>100</th>
<th>300</th>
<th>500</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>W</td>
<td>S</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>2.14</td>
<td>4.12</td>
<td>3.09</td>
</tr>
<tr>
<td>Sorted list (insert)</td>
<td>2.27</td>
<td>5.96</td>
<td>5.96</td>
</tr>
<tr>
<td>Merging sorted arrays</td>
<td>2.07</td>
<td>OOT</td>
<td>30.1</td>
</tr>
<tr>
<td>Binary tree (searches)</td>
<td>6.05</td>
<td>1043.68</td>
<td>1043.68</td>
</tr>
<tr>
<td>Dijkstra’s</td>
<td>26.36</td>
<td>27.69</td>
<td>OOT</td>
</tr>
<tr>
<td>Boolean matrix multiplication</td>
<td>928.69</td>
<td>2022.02</td>
<td>OOT</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Program</th>
<th>Time statistic</th>
<th>Scales</th>
<th>40</th>
<th>50</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion sort</td>
<td>Model building</td>
<td>9.28</td>
<td>9.28</td>
<td>9.28</td>
<td>9.28</td>
</tr>
<tr>
<td></td>
<td>Path prediction</td>
<td>0.32</td>
<td>0.06</td>
<td>0.06</td>
<td>0.06</td>
</tr>
<tr>
<td></td>
<td>Solver</td>
<td>0.10</td>
<td>0.15</td>
<td>0.15</td>
<td>0.15</td>
</tr>
<tr>
<td>Sorted list Insert</td>
<td>Model building</td>
<td>4.67</td>
<td>4.67</td>
<td>4.67</td>
<td>4.67</td>
</tr>
<tr>
<td></td>
<td>Path prediction</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>Solver</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>Merging sorted arrays</td>
<td>Model building</td>
<td>10.98</td>
<td>10.98</td>
<td>10.98</td>
<td>10.98</td>
</tr>
<tr>
<td></td>
<td>Path prediction</td>
<td>0.13</td>
<td>0.13</td>
<td>0.13</td>
<td>0.13</td>
</tr>
<tr>
<td></td>
<td>Solver</td>
<td>0.10</td>
<td>0.16</td>
<td>0.16</td>
<td>0.16</td>
</tr>
<tr>
<td>Dijkstra’s</td>
<td>Model building</td>
<td>25.53</td>
<td>25.53</td>
<td>25.53</td>
<td>25.53</td>
</tr>
<tr>
<td></td>
<td>Path prediction</td>
<td>1.92</td>
<td>1.15</td>
<td>1.15</td>
<td>1.15</td>
</tr>
<tr>
<td></td>
<td>Solver</td>
<td>0.26</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
</tr>
<tr>
<td>Boolean matrix multiplication</td>
<td>Model building</td>
<td>13.68</td>
<td>13.68</td>
<td>13.68</td>
<td>13.68</td>
</tr>
<tr>
<td></td>
<td>Path prediction</td>
<td>55.37</td>
<td>92.92</td>
<td>1102.13</td>
<td>1102.13</td>
</tr>
<tr>
<td></td>
<td>Solver</td>
<td>9.90</td>
<td>14.41</td>
<td>14.41</td>
<td>14.41</td>
</tr>
</tbody>
</table>
VIII. RELATED WORK

Analyzing the worst-case behavior or the worst case execution time (WCET) has been a subject of extensive research [27], [21], [17], [1], [19]. 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. The first-generation approach to deal with such a scalability issue is concolic execution [22], [29], which is a mix between concrete execution and symbolic execution. This can avoid prohibitive computational cost associated with full-blown symbolic execution at the cost of reduced coverage. Tools such as SAGE [16], Bouncer [10], KLEE [8], and Pex [31] handled loops by bounding the number of iterations but unable to reason about program behaviors beyond the loop bound.

Guiding path exploration is a natural approach to reduce execution overhead. WISE [7] utilizes full symbolic execution on small-scale tests to learn branch policies and applies the learned branching policy to generate large scale worst-case inputs. SPF-WCA takes a similar approach [24] with a more sophisticated history-based approach for learning branch patterns. Zhang et al. [37] proposed incremental symbolic execution that iteratively deepens exhaustive search depth in a branch tree, pruning out most of similar paths. This approach can automate program analysis and can be used for programs with much complex branch patterns. However, 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, XS TRESSOR invokes a constraint solver only once at a target large scale test, which is to solve out the input that satisfies constraints in the path that XS TRESSOR predicts to be the worst-case path.

Loop summarization is recent approach in multi-path loop analysis where the effect of a loop is summarized and reflected to the symbolic values at the exit of the loop [35]. LESE [28] 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 tools summarize loop effects well in small scale tests. However, they do not have any special consideration for large scale tests, thereby resulting in exponentially increasing analysis time according to input size. 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. PertFuzz [23] and SlowFuzz [26] use heuristics to drive input fuzzers to find long-running execution paths. Unlike XS TRESSOR (or WISE and SPF-WCA), they cannot provide any guarantees about finding worst-case behavior, as they are best-effort fuzzers.

Recently, Singularity [33] 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. Unlike XS TRESSOR, which models program behavior to find long-running paths, Singularity models input behavior to find long-running inputs. 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 XS TRESSOR, WISE, and SPF-WCA may not be able to model such a complex program. On the flip side, XS TRESSOR, 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 scaling 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 to stress test generation as the constraint solving entailed by symbolic analysis does not scale well to large-scale executions. Here we presented XS TRESSOR that solves this problem. It uses novel modeling techniques to build predictive models of what large-scale inputs will look given small scale input examples. 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 XS TRESSOR 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.

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

<table>
<thead>
<tr>
<th>Application</th>
<th>Model building time (seconds)</th>
<th>Prediction time(seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>W I X</td>
<td>50</td>
</tr>
<tr>
<td>GNU: chip</td>
<td>2.98 1.74 4.254</td>
<td>1.86 1.81 1.86</td>
</tr>
<tr>
<td>GNU: grep</td>
<td>10.99 OOT 22.70</td>
<td>OOT</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Application</th>
<th>Model building time (seconds)</th>
<th>Prediction time(seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>W I X</td>
<td>50</td>
</tr>
<tr>
<td>GNU: chip</td>
<td>2.98 1.74 4.254</td>
<td>1.86 1.81 1.86</td>
</tr>
<tr>
<td>GNU: grep</td>
<td>10.99 OOT 22.70</td>
<td>OOT</td>
</tr>
</tbody>
</table>