Harnessing Static Analysis to Help Learn Pseudo-Inverses of String Manipulating Procedures for Automatic Test Generation

Oren Ish-Shalom[1], Shachar Itzhaky[2], Roman Manevich[3], and Noam Rinetzky[1]

[1] Tel Aviv University, Israel
[2] Technion, Israel
[3] Ben-Gurion University of the Negev, Israel

Abstract. We present a novel approach based on supervised machine-learning for inverting String Manipulating Procedures (SMPs), i.e., given an SMP $p : \bar{\Sigma} \to \bar{\Sigma}$, we compute a partial pseudo-inverse function p^{-1} such that given a target string $t \in \bar{\Sigma}$, if $p^{-1}(t) \neq \bot$ then $p(p^{-1}(t)) = t$. The motivation for addressing this problem is the difficulties faced by modern symbolic execution tools, e.g., KLEE, to find ways to execute loops inside SMPs in a way which produces specific outputs required to enter a specific branch. Thus, we find ourselves in a pleasant situation where program analysis assists machine learning to help program analysis.
Our basic attack on the problem is to train a machine learning algorithm using (output, input) pairs generated by executing p on random inputs. Unfortunately, naively applying this technique is extremely expensive due to the size of the alphabet. To remedy this situation, we present a specialized static analysis algorithm that can drastically reduce the size of the alphabet Σ from which examples are drawn without sacrificing the ability to cover all the behaviors of the analyzed procedure. Our key observation is that often a procedure treats many characters in a particular uniform way: it only copies them from the input to the output in an order-preserving fashion. Our static analysis finds these *good* characters so that our learning algorithm may consider examples coming from a reduced alphabet containing a single representative good character, thus allowing to produce smaller models while using fewer examples than had the full alphabet been used. We then utilize the learned pseudo-inverse function to invert specific desired outputs by translating a given query to and from the reduced alphabet.
We implemented our approach using two machine learning algorithms and show that indeed our string inverters can find inputs that can drive a selection of procedures taken from real-life software to produce desired outputs, whereas KLEE, a state-of-the-art symbolic execution engine, fails to find such inputs.

1 Introduction

Recently, there has been a growing interest in applying machine learning techniques to challenging program analysis problems [9, 22, 23, 25, 27, 28, 32]. In this paper, we address the dual question: Can program analysis techniques help machine learning? We perform a preliminary case study in which machine learning algorithms are used to invert *string manipulating procedures* (SMPs), and show that in this domain the answer

is reassuringly positive. Interestingly, the models generated by the machine learning algorithms can themselves be of help to other program analysis tools. Specifically, they can help improve the coverage of symbolic execution tools such as KLEE [2]. Thus, we find ourselves in a pleasant situation where program analysis assists machine learning to help program analysis.

Research problem. Let Σ be a (possibly infinite) set of *characters* ranged over by a meta-character σ. A *string* $s \in \overline{\Sigma}$ is a finite sequence of *characters*. Given a deterministic SMP $p()$ which transforms input strings $s \in \overline{\Sigma}$ to output strings $p(s) \in \overline{\Sigma}$, where $p(s)$ denotes the output $p()$ returns when invoked on s, our goal is to find a *partial right pseudo-inverse* of a (possibly non-injective) p, i.e., a function $p^{-1} : \overline{\Sigma} \hookrightarrow \overline{\Sigma}$ such that

$$\forall s' \in \overline{\Sigma}.\, p^{-1}(s') \neq \bot \implies p(p^{-1}(s')) = s' .$$

Clearly, the problem is decidable as we can always have $p^{-1} = \bot$. Another trivial solution is to define p^{-1} to be the identity function wherever it coincides with the inverse of p, i.e., have

$$p^{-1}(s') = \begin{cases} s' & p(s') = s' \\ \bot & \text{otherwise} . \end{cases}$$

Thus, the challenge is to come up with a function p^{-1} with non-trivial domain of definition. Ideally, p^{-1} should be able to help automatic test generation, as we discuss now.

Motivation. The ability to invert string-manipulating procedures (SMPs) is useful, for example, in the context of tools for automatic test generation, e.g., KLEE [2]—a state-of-the-art symbolic execution engine. These tools automatically generate test cases, aiming to exercise as much of the program's code as possible. For example, KLEE uses various heuristics to explore the program's code: it continuously selects code paths that lead to not-yet-explored statements, applying a satisfiability-modulo theory solver (SMT) [4, 6] to determine whether a path is feasible, i.e., that there is an input which causes the selected path to be executed. As the exploration is path-sensitive, the tool may inspect an exponential number of code paths when exploring a loop containing a conditional, while generating formulae whose size is proportional to the length the inspected path. As a result, it can be challenging to cover a statement following a call to an SMP p which can be reached only if p returns a specific output s'. Ideally, when the engine reaches such a difficult-to-handle branch condition, one would want the symbolic execution engine to abandon the execution path it followed within $p()$, and instead, try to execute it "backward" to produce s'. Our technique equips the engine with such an ability by generating an "inverse shortcut"—a function that inverts the behavior of $p()$ without the cost of a path-by-path exploration.

Learning pseudo-inverses. Our goal is to help tools such as KLEE to find inputs which drive SMPs to produce desired outputs. We suggest to do it using machine learning: Given an SMP $p()$ mapping input strings s to output strings $p(s)$, we apply a supervised machine learning algorithm to learn a *model* of a pseudo inverse of p. The model should

be capable of *translating* strings, i.e., given a string s' the model should be able to find a string s which it predicts to be an inverse of s under p.

Roughly speaking, producing the model entails generating a set of arbitrary inputs $\{s_i\}_{i=1}^n$, executing p on each input, thus producing a training set $T = \{(p(s_i), s_i)\}_{i=1}^n$, and finally training the algorithm on T. Note that T is comprised of pairs of strings mapping the *output* of $p()$ to the input which produced it. Thus, by training the algorithm using T, we in fact learn a model which approximates the behavior of a pseudo-inverse of $p()$.

The challenge. Unfortunately, a naive generation of the training set can be extremely inefficient in the sense that many output/input pairs effectively expose the same behavior. For example, consider an SMP which adds an escape character before tab and newline characters in its input. If we use randomly generated training sets, $p()$ will act as the identity function on most of the examples, and it might require a very large training set to expose other, more "interesting", behaviors: A randomly constructed string with 10 resp. 88 characters has a 92% resp. 50%, chance not to include a tab or a newline character. As a result, the machine learning algorithm might find it difficult to generalize the interesting cases (or outright ignore them, considering them to be noise), and end up learning a bad approximation of the inverse.

Our solution: Learning with reduced alphabets. To remedy the above situation, we propose a static analysis which allows to reduce the alphabet from which the training set examples are drawn, without scarifying the ability to encode any "interesting" behaviors. In fact, our approach increases the chances of generating "interesting" examples by reducing the part of the alphabet from which "non-interesting" examples are drawn. Intuitively, we identify a *set of good characters* (Definition 2) whose only effect on the analyzed procedure is to be copied in an order-preserving manner from the input to the output. Our *key insight* is that given such a set, it is possible to expose all the interesting behaviors of a procedure using an alphabet containing a single representative good character, and deduce the effect of an SMP on a string containing characters which were not found in the reduced alphabet from its effect on a *similar* string (Definition 1) whose characters do.

Alphabet reduction via static analysis. To automate the selection of good characters, we designed a static analysis that can find the set of good characters for a given string manipulating procedure. Our analysis handles a restricted class of procedures. In this class, a procedure takes a string input and returns a string output. The procedure can read its input from left-to-right, from right-to-left, or in both directions, however each input character is read only once. The procedure is allowed to use variables that can hold character values and employ conditionals and loops, where condition can only test whether a character variable is equal or not to another variable or to a constant. While simple, we found that our restricted programming model is still expressive enough to handle a variety of procedures.

Technically, the static analysis maintains an order between the variables according to the position of the input character that they got their value from. Essentially, whenever a variable x containing an input character is written to the output out of turn, i.e.,

before any other variable y holding an input value σ which was read off the input x was set, the analysis determines that the σ cannot be good. Similarly, writing a constant character `const` to the output leads the analysis to dictate that `const` is not good either.

Implementation and experimental evaluation. We implemented our analysis in a tool called StrInver. We applied it to invert a small selection of procedures written in *C* and taken from real-life software. (The tool operates on LLVM bitcode.) We then ran KLEE on a simple program containing a call to the SMP followed by an erroneous command whose execution is predicated on the SMP returning a particular output. Our analysis succeeded to find useful pseudo-inverses of the particular outputs in a few seconds, whereas KLEE, a state-of-the-art symbolic execution tool, failed to find an input which lead to the bug.

Main contributions. The main conceptual contribution of our work is the observation that when a machine learning algorithm is used to discover properties of programs, it might be possible to use program analysis to help direct the choice of the training set towards examples that expose interesting behaviors. The main technical contribution of our work is the concretization of this observation by developing a static analysis algorithm which allows to reduce the size of the alphabet from which examples are drawn when learning pseudo inverses of a restricted class of string manipulating procedures. The main practical contribution of our work is the implementation of the analysis and an empirical evaluation where we applied our technique to a small selection of procedures taken from real-life software. Also, to the best of our knowledge, the idea of using machine learning to invert string-manipulating procedures is novel.

2 Overview

In this section, we motivate our research problem and give a high-level overview of our approach by walking the reader through a series of examples.

Example 1 *Figure 1 shows procedure* `escapeWS()`*, an SMP which returns a copy of its input string with a* \$ *character before every* 5 *and* 8 *character it contains.*[4] *For example, given the input string "*`Ali5BaBa8`*",* `escapeWS()` *outputs "*`Ali$5BaBa$8`*".*

To motivate the need for computing inverses of SMPs, assume that we wish to symbolically execute a program which aborts in an error state only if `escapeWS()` produces a particular output, e.g.,

```
ret = escapeWS(input);
if (strcmp(ret, "Ali$5BaBa$8") == 0) abort();
...
```

[4] The procedure is based on a GCC procedure which adds an escape character before tab and newline characters. For clarity, we replaced the whitespace characters with more visible characters. For simplicity, we removed code concerning array bound checking.

```
char* escapeWS(char *in) {
  char *out = malloc(MAX_STRLEN),
       *s = out;
  while (*in != 0) {
    if (*in == '5' || *in == '8') {
      *s = '$'; s++ }
    *s = *in; in++; s++ }
  *s = *in;
  return out }
```

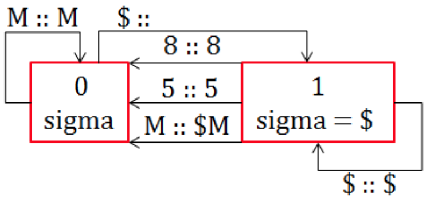

Fig. 1: A simple SMP[4] and a transducer approximating its pseudo-inverse under the reduced alphabet

Note that `escapeWS()` produces "`Ali$5BaBa$8`" only if it is given "`Ali5BaBa8`" as input. To find this input, symbolic execution engines such as KLEE would have to follow a very particular code path, namely the one in which the loop body is executed nine times and the true branches of the first and second `if` statements are taken after reading the fourth and ninth input characters, respectively.

Our goal is to help tools such as KLEE to find inputs which drive SMPs to produce specific desired outputs. We would like to use an off-the-shelf supervised machine learning algorithm and train it to generate a model of the inverse function. While it is quite easy to generate random inputs, most of them will be non-representative of the function's actual semantics, necessitating large training sets, as we noticed in our experiments. Consider again procedure `escapeWS()` shown in Figure 1. It is easy to see that the procedure acts rather uniformly on most of the input characters: all the characters are copied from the input string to the output string in an order-preserving fashion, only characters 5 and 8 trigger an insertion of the '$' character. Thus, if the input string does not contain characters 5 and 8 then the procedure acts as the identity function. Thus, intuitively, all the "interesting" behaviors of the procedure should be detected by considering string comprised of four characters: 5, 8, $, and an arbitrary character *M* representing all other characters.

Inverting SMPs with reduced alphabets. To remedy the above situation, we propose a static analysis which allows to reduce the alphabet from which the training set examples are drawn, without scarifying the ability to encode any "interesting" behaviors. Our *key insight* is that if we can identify that the SMP does not distinguish between several characters then it might be possible to expose all the interesting behaviors of a procedure using an alphabet containing a single representative character of the set, and deduce the effect of an SMP on a string containing characters which were not found in the reduced alphabet from its effect on a *similar* string whose characters do. In this paper, we focus on a particular class of indistinguishable characters, those which the procedure act on as, essentially, the identity function, and refer to these characters *good* characters.

Definition 1 (Similar strings) *Let $S \subseteq \Sigma$ be a set of characters. Strings $s_1 \in \overline{\Sigma}$ and $s_2 \in \overline{\Sigma}$ are* similar up to *S, denoted by $s_1 \sim_S s_2$, if $|s_1| = |s_2|$, where $|s|$ denotes the length of a string s, and for every $i = 1..|s_1|$, it holds that either $s_1(i) = s_2(i)$ or $\{s_1(i), s_2(i)\} \subseteq S$.*

Definition 2 (Good characters) *Given a procedure $p()$ and a string s, a set of characters $G(s) \subseteq \Sigma$ is* good *for s and $p()$ if $s|_{G(s)} = p(s|_{G(s)})$, where $s|_{G(s)}$ denotes the maximal subsequence of s comprised of characters in $G(s)$. A set of characters G is* good *for $p()$ if it is good for $p()$ and any input string s.*

Lemma 1. *Let $G \subseteq \Sigma$ be a set of good characters for $p()$. For any two strings s_1 and s_2, if $s_1 \sim_G s_2$ then $p(s_1) \sim_S p(s_2)$.*

Given a procedure $p()$, our static analyzer, discussed next, finds a set of good characters for $p()$ by way of elimination. For example, our analyzer finds that the set $\{\$, 5, 8\}$ is bad for procedure `escapeWS()`. We use this result to construct a *reduced alphabet* $\Gamma = \{\$, 5, 8, M\}$, where $M \in \Sigma$ is the single representative of the good characters, which we refer to as a *metacharacter*. Given Γ, we apply the aforementioned learning process; this time, however, we generate the training set by only drawing examples from Γ. Our static analysis is independent of the machine learning algorithm used to find the inverse. In our experiments, we use two such algorithms: OSTIA [14], which learns a *transducer*, and the other is a non deterministic model for character insertion/replacement/deletion based on the Needleman Wunsch alignment algorithm [20]. (See Section 6.) A transducer is a finite state machine that, instead of accepting or rejecting an input string, outputs characters upon transition. Figure 1 depicts a transducer approximating the pseudo inverse of `escapeWS()` which OSTIA has learned using a training set comprised of 100 strings, randomly generated over Γ. (We explain the graphical notations in Section 6.1.) In our example, the transducer has two states, where state 0 is the initial one. An edge labeled $\xrightarrow{x::s}$ is traversed when reading an input character x and it outputs the string s. (For further details, see Section 6.) For example, when applied to the string $s' = MM\$5\$5MM\$8$, the transducer outputs the string $s = MM55MM8$. Note that executing `escapeWS()` on s results in s', i.e.,

$$escapeWS^{-1}(MM\$5\$5MM\$8) = MM55MM8.$$

In fact, if we only consider strings comprised of characters coming from Γ then the transducer in Figure 1 can invert any string in the image of `escapeWS()`.

Static analysis for a restricted class of SMPs. One of the key technical contributions of this paper is the design of a static analysis that can find a set of good characters for a given string manipulating procedure. Our analysis handles a restricted class of procedures. In this class, a procedure takes a string input and returns a string output. The procedure can read its input from left to right or from right to left, however each input character is read only once. The procedure is allowed to use variables that can hold character values and employ loops and conditions where a variable is compared with a constant character or another variable. While simple, we found that our restricted programming model is still expressive enough to handle a variety of procedures.

The analysis abstracts the execution trace of the procedure by maintaining an ordering over program variables according to the position of the input character that they got their value from. Essentially, whenever a variable containing an input character is written out of order, the analysis determines that the values of all the unwritten variables that may have been read before it are *not* good. Writing a constant character to the output also leads the analysis to decide that this character is not good either.

String inversion. Given the machine learning model approximating a pseudo inverse of $p()$ and an output string s', we first replace every good character in s' with the meta character. For example,

$$s' = Ab\$5\$5T@\$8 \xrightarrow{\text{translates to}} s'_0 = MM\$5\$5MM\$8\,.$$

We then execute the transducer using s'_0, which returns s_0. Recall that $escapeWS^{-1}(s'_0) = s_0$. Our main theorem (see Section 5) ensures that the static analysis indeed finds a set of good characters for the analyzed procedure. Thus, using Definition 2, we can get an input s which would lead to the desired output s' by replacing the meta character M back to the good character it came from in s'. For example,

$$s_0 = MM55MM \xrightarrow{\text{translates back to}} s = Ab55T@8.$$

Indeed, $escapeWS(Ab55T@8) = Ab\$5\$5T@\$8$.

Disclaimer. We note that our technique is not guaranteed to always find an input s which leads $p()$ to produce a particular output string s'. This can be because $p()$ is not surjective and $p^{-1}(s')$ is undefined, or because the translation model produced by the machine learning algorithm is not accurate enough. (To isolate any client application from this kind of inaccuracy, and as we have $p()$ at our disposal, we can execute $p()$ on s and validate that indeed it returns s'). Furthermore, in case $p()$ is not injective, and there might be multiple inputs leading to a specific output, the model we learn may return an arbitrary string, which, untested by a forward execution, might not even be in the preimage of p. However, as our technique involves generating a random training set, re-executing the learning phase may create a different model which would possibly find a different input.

3 Programming Language

We formalize our results for a simple imperative programming language specialized for string processing: Every program receives a string as input and produces a string as output. The design of the language is inspired by real life examples of string manipulating procedures which often process their inputs character by character.[5]

[5] We remind the reader that our tool operates directly on LLVM bitcode.

$$
\begin{array}{rl}
\mathit{stmt} & ::= \mathit{cmd} \mid \mathit{stmt}\ \mathit{stmt} \mid \\
& \quad \mid \texttt{if}\ (\mathit{bexp})\ \{\ \mathit{stmt}\ \}\ \texttt{else}\ \{\ \mathit{stmt}\ \} \\
& \quad \mid \texttt{while}\ (\mathit{bexp})\ \{\ \mathit{stmt}\ \} \\
\mathit{cmd} & ::= x = \texttt{read-first()} \mid x = \texttt{read-last()} \\
& \quad \mid \texttt{write-first}(x) \mid \texttt{write-last}(x) \\
& \quad \mid x := \mathit{exp} \mid \texttt{return} \\
\mathit{exp} & ::= \texttt{const} \mid y \\
\mathit{bexp} & ::= x \bowtie \mathit{exp} \mid \texttt{done()} \mid \texttt{!done()}
\end{array}
$$

```
d = $
while(!done()) {
    x = read-first()
    if (x = 5) {
        write-first(d) }
    if (x = 8) {
        write-first(d) }
    write-first(x) }
return
```

Fig. 2: Syntax of the programming language and a version of procedure `escapeWS()` written in our programming language. $\bowtie \in \{=, \neq\}$

Computation model. Roughly speaking, programs have at their disposal two *read heads* and two *write heads*. The input resides in the *read buffer*. The *first read head* is used to read the input from left to right, and the *last read head* allows to read the input from right to left. Once a read head inspects a character, it advances to the next position. A special built-in expression `done()` allows the programmer to determine whether all the input characters have been read. Trying to read a character after all the input has been read blocks the program.[6] The program produces its output using the write heads. The *first write head* writes characters to the program's *first write buffer*, and the *last write head* writes to the program's *last write buffer*. The first write buffer is written from left to tight and the last write buffer is written from right to left. When the program terminates, it returns a concatenation of the first and last write buffers. This model allows us to handle programs which process their input string in a character by character fashion and read every character at most once in a sequential manner going from the beginning of the string to it end, the other way around, or even alternating between the two directions.

3.1 Syntax

Figure 2 defines the syntax of our programming language and, as an example, shows a possible encoding of procedure `escapeWS()` in our language.

A program is a statement *stmt*, which can be either a primitive command *cmd*, a sequential composition of statements, denoted by juxtaposition, a conditional statement, or a while loop.

Primitive commands allow to read input characters, write output characters, and assign the values of *expressions* to variables: The command $x =$ `read-first()` reads a character from the input using the first reading head, and assigns it to variable x. The command $x =$ `read-last()` does the same using the last read head. The commands `write-first(x)` and `write-last(x)` write the contents of x to the first and last output buffers, respectively. We allow for assignments of the form expression $x = \mathit{exp}$ where an expression *exp* is either a character variable or a constant character. The `return` command terminates the execution of the program and produces the output by concatenating the write buffers.

[6] The choice to block the program was done in the sake of simplicity. Alternatively, we could have designed the language to signal an error.

$$(\sigma in, out_F, out_L, env) \xrightarrow{v=read-first()} (in, out_F, out_L, env[v \mapsto \sigma])$$
$$(in, out_F, out_L, env) \xrightarrow{write-first(v)} (in, out_F\, env(v), out_L, env)$$
$$(in\sigma, out_F, out_L, env) \xrightarrow{v=read-last()} (in, out_F, out_L, env[v \mapsto \sigma])$$
$$(in, out_F, out_L, env) \xrightarrow{write-last(v)} (in, out_F, env(v)\, out_L, env)$$
$$(in, out_F, out_L, env) \xrightarrow{v:=\texttt{const}} (in, out_F, out_L, env[v \mapsto \texttt{const}])$$
$$(in, out_F, out_L, env) \xrightarrow{v:=x} (in, out_F, out_L, env[v \mapsto env(x)])$$
$$(in, out_F, out_L, env) \xrightarrow{assume(x \bowtie \texttt{const})} (in, out_F, out_L, env) \qquad env(x) \bowtie \texttt{const}$$
$$(in, out_F, out_L, env) \xrightarrow{assume(x \bowtie y)} (in, out_F, out_L, env) \qquad env(x) \bowtie env(y)$$
$$(in, out_F, out_L, env) \xrightarrow{assume(done())} (in, out_F, out_L, env) \qquad in = \varepsilon$$
$$(in, out_F, out_L, env) \xrightarrow{assume(!done())} (in, out_F, out_L, env) \qquad in \neq \varepsilon$$
$$(in, out_F, out_L, env) \xrightarrow{return} out_F\, out_L$$

Fig. 3: Concrete meaning of commands. $\bowtie \in \{=, \neq\}$

Conditional statements and while loops use boolean expressions *bexp* which allow to check whether the value of a given variable is equal to a given expression or not. Two additional boolean expressions are the special built in operators `done()` and `!done()`, which allow the program to determine whether all its input has, respectively, has not, been read.

3.2 Concrete Semantics

Before defining the meaning of programs in our language, we introduce some notation.

Notation. We assume a (possibly infinite) domain (alphabet) of characters Σ ranged over by the meta variable σ, and a syntactic domain *VAR* of variable names, ranged over by $v, x, \ldots$, which we also use as a semantic domain. A *string* $s \in \overline{\Sigma}$ is a sequence of characters, i.e., a function from $1..n$, for some $n \in \mathbb{N}^0$, to Σ. In the following, we denote string concatenation by juxtaposition and the empty string by ε. Given a function *env*, we write $env[x \mapsto y]$ to denote a function which acts like *env* anywhere except at x, where its value is y.

Concrete memory states. A *concrete memory state*

$$m = (in, out_F, out_L, env) \in \mathcal{M} = \overline{\Sigma} \times \overline{\Sigma} \times \overline{\Sigma} \times E$$

is a quadruple. The first three components, namely *in*, out_F, and out_L, are strings which store the contents of the program's read buffer, first write buffer, and last write buffer, respectively. $env \in E = VAR \to \Sigma$ is an *environment* which records the values of variables. We assume that variables are initialized to some fixed *zero* character. By abuse of notation, we let $env(\texttt{const}) = \texttt{const}$ for any constant character `const`.

Operational semantics. Figure 3 defines the meaning of programs using a small step operational semantics. The latter is defined by translating the program into a control-flow graph form, and encoding conditional using `assume` commands in the standard way: executing an `assume(bexp)` command blocks the execution on state where `bext` does not hold, and does not change the state otherwise. The meaning of commands is rather self explanatory. We only direct the reader's attention to the fact that a `write-first()` command adds a character at the *end* of the first write buffer whereas the `write-last()` command adds a character at the *beginning* of the last write buffer and that the program gets stuck if it tries to read an input character when the read buffer is empty.

4 Instrumented Semantics

The purpose of our static analyzer is to help reduce the size of the input alphabet used by the machine learning algorithm when computing the pseudo-inverse of the analyzed SMP. To do so, it detects characters on which the SMP act as the identity function: It turns out that rather often a string-manipulating procedure treats many characters in a particularly uniform way; it only copies them once from the input to the output in an order-preserving fashion. The static analyzer conservatively finds these *good* characters, and enables the use of a single good representative character in the alphabet during learning. This reduction in the size of the alphabet translates to a huge benefit for the learning algorithms, as we discovered in our experiments.

The instrumented semantics extends the concrete one with properties which are of matter to the analysis. The main tracked property is the set $BAD \subseteq \Sigma$ of *bad* characters for the execution. We explain the role of BAD by describing its complement $GOOD = \Sigma - BAD$. A set of characters $G \subseteq \Sigma$ is *good* if every time a character $\in G$ is read off the input, it is copied as-is to the output. In particular, the subsequences of the input and output strings comprised of the good characters are identical. (see Definition 2.)

The goal of our static analysis is to determine a set of good characters for an SMP. The role of the instrumented semantics is to explicate which properties of the execution are tracked to facilitated this task. Thus, when the instrumented semantics terminates, it returns, in addition to the output string, the set of bad characters it computed. Let $\hat{p}(s) = (s', BAD)$ denote the output $p()$ produces if it executes according to the instrumented semantics on input string s. The usefulness of the instrumented semantics as a basis for analysis stems from the following lemma.

Lemma 2. *Let $p()$ be an SMP and s, s' strings. If $\hat{p}(s) = (s', BAD)$ then $\Sigma \setminus BAD$ is a* good *set of characters for $p()$ and s.*

4.1 Instrumented States

Instrumented states record properties pertaining to the flow of information from the input string through variables to the output string. Specifically, every instrumented state augments a concrete state with four binary relations $E_F, E_L, R_F, R_L \subseteq \Sigma \times \Sigma$ and the set of possibly-not-good characters $BAD \subseteq \Sigma$. We refer to the quintuple $\iota = (E_F, E_L, R_F, R_L, BAD)$

```
   x = read-first()
   y = read-first()
3: z = y
   if (y = $) {
      write-first(y)
      write-first(x)
7: }
   else {
      write-first(x)
      write-first(z)
   }
   while (!done()) {
      x = read-first()
      write-first(x) }
```

Fig. 4: SMP `showDough()`

as an *instrumentation*. We assume the components of the instrumentation are initialized to $\emptyset$.

$$\hat{m} = (m, \iota) \in \hat{\mathcal{M}} = \mathcal{M} \times \hat{I}$$
$$\text{where } \iota = (R_F, R_L, E_F, E_L, BAD) \in \hat{I} = (2^{VAR \times VAR})^4 \times 2^{\Sigma}.$$

The m component of an instrumented state is the concrete state it augments.

E_F and E_L are equivalence relations over variable names. Recall that in our language, a variable can be assigned a value either by reading into it a character from the input, assigning into it a constant value, or assigning into it the value of another variable. E_F equates variables whose values originated from the same `read-first()` operation, either directly, or through a sequence of copy assignments. For example, in the instrumented state which arises at program point 3 in Figure 4, $E_F = \{(x,x),(y,y),(y,z),(z,y),(z,z)\}$. E_L does the same for variables whose value was originated from a `read-last()` command. In Figure 4, there are no `read-last()` commands. Thus, $E_L = \emptyset$ at any state which arises during the execution.

R_F and R_L are preorders over variable names. R_F represents the order in which variables were read using the first reading head, and R_F represents the order in which variables were read using the last reading head. Both orders take variable copy-assignments into account. For instance, at the instrumented state in program point 3, $R_F = \{(x,x),(y,y),(z,z),(y,z),(z,y),(x,y),(x,z)\}$. $R_L = \emptyset$ because there are no `read-last()` commands.

BAD over approximates the set of bad characters for the input string on which the SMP executes to produce the state. The over approximation is based on the flow of characters from the input string to output string, as we discuss in Section 4.2.

Healthiness conditions. The instrumentation in instrumented states respects certain natural *healthiness* conditions: A variable may appear in R_F only if it appears in E_F, as in a concrete execution the order in which input characters is read is total and every input character may be read at most once. In fact, R_F can be seen as a total order over the

equivalence classes of E_F. A similar relation exists between R_L and E_L. Finally, a variable cannot appear in both E_F and E_L as an input character may be read either by the first read head or by the last read head.

4.2 Instrumented Small-Step Operational Semantics

The instrumentation is manipulated by the instrumented transformers presented in Figure 5 which defines a deterministic transition relation over $\hat{I} \times \hat{I}$.[7] The transition rules of the instrumented semantics extends the ones of the concrete semantics to track *must* value-flow information:

$$\frac{m \xrightarrow{cmd} m' \quad \iota \xrightarrow{cmd}_m \iota'}{(m,\iota) \xrightarrow{cmd} (m',\iota')}\ cmd \neq \texttt{return} \qquad \frac{m \xrightarrow{\texttt{return}} s \quad \iota \xrightarrow{\texttt{return}}_m BAD}{(m,\iota) \xrightarrow{\texttt{return}} (s,BAD)}$$

The transition relation of the instrumented part of the state is parameterized with the source concrete state of the transition because it requires access to the environment.

We define the rules in Figure 5, which we explain next, using the following shorthand: Let R resp. E be a binary resp. equivalence relation over variable names and X a set of variable names. We use the following as shorthand $R|_{\neg X} \equiv \{(a,b) \in R \mid (a \notin X) \wedge (b \notin X)\}$ removes from R any pair coming from $X \times X$, $[\![v]\!]_E \equiv \{x \mid (x,v) \in E\}$ denotes the equivalence class of v in E, and $\text{Add}(R,v,x) \equiv R \cup \{(a,v) \mid (a,x) \in R\} \cup \{(v,a) \mid (x,a) \in R\}$ adds v to R in the same positions as x.

The instrumented semantics of a $v = \texttt{read-first}()$ command removes any mention of v from all the relations in the instrumentation—it might be there because its value could have come from a previous read command. It then places it in its own equivalence class in E_F and as the the minimal element in R_F: v is the only variable that got its value from that `read-first()` operation, which is the last command executed so far. If before the assignment v relates only to itself in either E_F or E_L then its value is about to be overridden and lost before having a chance to get written to the output. Hence, in this case the value of $env(v)$ is considered a bad character.

The instrumented meaning of $\texttt{write-first}(v)$ removes any mention of v or any of the variables in its equivalence class according to E_F from any relation it belongs to. This is because a *read* good character should not be written more than one time to the output. If v got its value from a constant assignment or from the opposite read head, or if its value has already been written then $env(v)$ becomes a bad character. If v did get its value from the *first-head* but it is not written in the right order, i.e., it is not a maximal element in R_F then the contents of all the larger variables in R_F becomes bad.[8]

The instrumented meaning of a $\texttt{v} := exp$ removes v from its current place in the instrumentation and, if exp is a variable, places v in the same relations and at the same positions as exp. If v was the only variable to contain a value coming from a read command then $env(v)$ becomes a bad character.

[7] The transformers pertaining to `read-last()` and `write-last()` operations are similar to those of `read-first()` and `write-first()`, respectively, and are thus omitted.

[8] An equally plausible alternative would be to make $env(v)$ bad.

$$(R_F,R_L,E_F,E_L,BAD) \xrightarrow{v=read-first()}_m (R^r_F,R_L|_{\neg v},E_F|_{\neg v}\cup\{(v,v)\},E_L|_{\neg v},BAD\cup BAD^r)$$

$$R^r_F = R_F|_{\neg v}\cup\{(x,v)\mid (x,x)\in E_F\}\cup\{(v,v)\}$$

$$BAD^r = \begin{cases}\{env(v)\} & [\![v]\!]_{E_F}=\{v\}\vee[\![v]\!]_{E_L}=\{v\}\\ \emptyset & otherwise\end{cases}$$

$$(R_F,R_L,E_F,E_L,BAD) \xrightarrow{write-first(v)}_m \left(R^w_F,R_L|_{\neg v},E_F|_{\neg([\![v]\!]_{E_F})},E_L|_{\neg([\![v]\!]_{E_L})},BAD\cup BAD^w\right)$$

$$R^w_F = \begin{cases}\{(z,y)\in R_F\mid z\notin[\![v]\!]_{E_F}\} & (v,v)\in E_F\\ R_F & otherwise\end{cases}$$

$$BAD^w = \begin{cases}\{env(v)\} & (v,v)\notin E_F\vee(v,v)\in E_L\vee exp=\texttt{const}\\ \{\sigma\in\{env(y)\}\mid (y,y)\in E_F\wedge y\neq v\wedge(v,y)\notin R_F\} & (v,v)\in E_F\wedge exp=v\end{cases}$$

$$(R_F,R_L,E_F,E_L,BAD) \xrightarrow{v:=\texttt{const}}_m (R_F|_{\neg v},R_L|_{\neg v},E_F|_{\neg v},E_L|_{\neg v},BAD\cup BAD_v)$$

$$(R_F,R_L,E_F,E_L,BAD) \xrightarrow{v:=x}_m (R_F|_{\neg v},R_L|_{\neg v},E_F|_{\neg v},E_L|_{\neg v},BAD\cup BAD_v) \qquad (x,x)\notin E_F\cup E_L$$

$$(R_F,R_L,E_F,E_L,BAD) \xrightarrow{v:=x}_m (\mathrm{Add}(R_F|_{\neg v},v,x),R_L,\mathrm{Add}(E_F|_{\neg v},v,x),E_L,BAD\cup BAD_v) \qquad (x,x)\in E_F$$

$$(R_F,R_L,E_F,E_L,BAD) \xrightarrow{v:=x}_m (R_F,\mathrm{Add}(R_L|_{\neg v},v,x),E_F,\mathrm{Add}(E_L|_{\neg v},v,x),BAD\cup BAD_v) \qquad (x,x)\in E_L$$

$$\text{where } BAD_v = \begin{cases}\{env(v)\} & [\![v]\!]_{E_F}=\{v\}\vee[\![v]\!]_{E_L}=\{v\}\\ \emptyset & otherwise\end{cases}$$

$$(R_F,R_L,E_F,E_L,BAD) \xrightarrow{return}_m BAD$$

Fig. 5: Instrumented semantics. The transformers pertaining to `assume` commands act like the identity function. $m=(\_,\_,\_,env)$. We assume $v\neq x$

The instrumented meaning of a `return` command ends the execution with the accumulated *BAD* set.

The rules in Figure 5 never interfere with neither the values nor the control of the underlying concrete semantics. They also preserve healthiness.

Lemma 3. *Let $(m,\mathfrak{l})$ and $(m',\mathfrak{l}')$ be instrumented states and cmd a command such that $(m,\mathfrak{l})\xrightarrow{cmd}(m',\mathfrak{l}')$. If $\mathfrak{l}$ is a healthy instrumentation then $\mathfrak{l}'$ is healthy too.*

5 Static Analysis

Our abstract interpretation algorithm over-approximates the instrumented semantics described in Section 4 by replacing the concrete memory state component of instrumented states with an abstract one.

Abstract States Our static analysis algorithm computes an *abstract instrumented state*

$$A=(m^\sharp,\mathfrak{l})\in\mathcal{A}=\mathcal{M}^\sharp\times\hat{\mathcal{I}} \qquad \text{where } m^\sharp=(Done,env^\sharp)\in\mathcal{M}^\sharp.$$

at every program point. An abstract instrumented state is comprised of an *abstract state* $m^\sharp = (Done, env^\sharp)$ and an instrumentation $\iota \in \hat{I}$ (see Section 4.1). The *Done* component of the abstract state abstracts the number of unread characters, i.e., whether the two read-heads passed each other or not: $\{T\}$ means that all the input characters have been read, $\{F\}$ means the opposite, and $\{T,F\}$ means that the situation is unknown. $env^\sharp : VAR \to 2^\Sigma$ is an abstract environment mapping variable names to the sets of their possible values. The instrumentation component ι is utilized for the same purpose and in the same way as in the instrumented semantics.

Notice that while $env^\sharp(x) \in 2^\Sigma$ may be infinite, the only changes to it are additions and removals of values that occur as literal constants in the program. Therefore the number of such distinct values is at most 2^k, where k is the number of such constants. This provides a termination guarantee of the analysis even with an infinite alphabet.

Join. The least upper bound (join) operator is defined as follows:

$$
\begin{aligned}
&(m_1^\sharp, \iota_1) \sqcup (m_2^\sharp, \iota_2) = (m_1^\sharp \sqcup m_2^\sharp, \iota_1 \sqcup \iota_2), \qquad \text{where} \\
&\quad (Done_1, env_1^\sharp) \sqcup (Done_1, env_1^\sharp) = (Done_1 \cup Done_1, \lambda x.\, env_1^\sharp(x) \cup env_2^\sharp(x)) \\
&\quad \left(R_F^1, R_L^1, E_F^1, E_L^1, C^1, BAD^1\right) \sqcup \left(R_F^2, R_L^2, E_F^2, E_L^2, C^2, BAD^2\right) = \\
&\qquad\qquad (R_F^1 \cap R_F^2, R_L^1 \cap R_L^2, E_F^1 \cap E_F^2, E_L^1 \cap E_L^2, BAD^3) \\
&\quad \text{where } \; BAD^3 = BAD^1 \cup \{\sigma \in \rho_1(x) \mid x \in E_F^1 - E_F^2 \cup E_L^1 - E_L^2\} \cup \\
&\qquad\qquad BAD^2 \cup \{\sigma \in \rho_2(x) \mid x \in E_F^2 - E_F^1 \cup E_L^2 - E_L^1\}
\end{aligned}
$$

With the exception of the *BAD* component, it is easy to see that the the resulting state is indeed the least upper bound of the two abstract instrumented states. The reason we chose in to intersect most of the component of joined instrumentations is rather clear—we track must information. To understand the reason why defining $BAD^3 = BAD^1 \cup BAD^2$ would not suffice to ensure a sound analysis consider a scenario when $(x,x) \in E_F^1 - E_F^2$. Had we kept $(x,x) \in E_F^3$, then a future `write-first`(x) possibly violates the goodness of the character set $\rho_3(x)$ as it may be written without ever being read. On the other hand, as we discarded (x,x) from E_F^3, we opened the door for a future `x = read-first()` to possibly violate the goodness of the character set $\rho_3(x)$, as some characters may have been read without ever being written. So whenever $(x,x) \in E_F^i - E_F^j$ ($\{i,j\} = \{1,2\}$) we include $\rho_i(x)$ in BAD^3. The same line of reasoning applies to E_L too. Thus, we add to BAD^3 the characters associated with the variables found in the symmetrical difference of the relevant equivalence realtions.

5.1 Concretization

The concrete domain which we use to justify the soundness of our analysis is the powerset of instrumented states. The concretization function γ maps an abstract state to a set of instrumented ones. Let $\iota = (R_F, R_L, E_F, E_L, BAD)$, then

$$
\begin{aligned}
\gamma(((Done, env^\sharp), \iota)) = \{((in, out_F, out_L, env), (R_F^c, R_L^c, E_F^c, E_L^c, BAD^c)) \mid \\
in = \varepsilon \to T \in Done \wedge in \neq \varepsilon \to F \in Done \wedge \\
\forall x.\, env(x) \in env^\sharp(x) \wedge \\
R_F^c \supseteq R_F \wedge R_L^c \supseteq R_L \wedge E_F^c \supseteq E_F \wedge E_L^c \supseteq E_L \wedge BAD^c \subseteq BAD
\end{aligned}
$$

$$\left(env^\sharp, Done\right) \xrightarrow{v=read-first()\ /\ v=read-last()} \left(env^\sharp[x \mapsto \Sigma], Done \cup \{T\}\right) \qquad Done \neq \{T\}$$

$$\left(env^\sharp, Done\right) \xrightarrow{write-first(v)\ /\ write-first(v)} \left(env^\sharp, Done\right)$$

$$\left(env^\sharp, Done\right) \xrightarrow{v:=\texttt{const}} \left(env^\sharp[v \mapsto \{\texttt{const}\}], Done\right)$$

$$\left(env^\sharp, Done\right) \xrightarrow{v:=x} \left(env^\sharp[v \mapsto env^\sharp(x)], Done\right)$$

$$\left(env^\sharp, Done\right) \xrightarrow{assume(v=\texttt{const})} \left(env^\sharp[v' \mapsto \{\texttt{const}\} \mid \varphi(v)], Done\right) \qquad \texttt{const} \in env^\sharp(x)$$

$$\left(env^\sharp, Done\right) \xrightarrow{assume(v!=\texttt{const})} \left(env^\sharp[v' \mapsto env^\sharp(x) \setminus \{\texttt{const}\} \mid \varphi(v)], Done\right) \qquad \{\texttt{const}\} \neq env^\sharp(x)$$

$$\left(env^\sharp, Done\right) \xrightarrow{assume(v=x)} \left(env^\sharp[v', x' \mapsto env^\sharp(v) \cap env^\sharp(x) \mid \varphi(v) \wedge \varphi(x)], Done\right) \qquad env^\sharp(v) \cap env^\sharp(x) \neq \emptyset$$

$$\left(env^\sharp, Done\right) \xrightarrow{assume(v!=x)} \left(env^\sharp[v \mapsto V, x \mapsto X \mid \varphi(v) \wedge \varphi(x)], Done\right) \qquad env^\sharp(v) = env^\sharp(x)$$
$$V = \text{if } (|env^\sharp(x)| = 1) \text{ then } env^\sharp(v) \setminus env^\sharp(x) \text{ else } env^\sharp(v) \qquad \Longrightarrow |env^\sharp(v)| > 1$$
$$X = \text{if } (|env^\sharp(v)| = 1) \text{ then } env^\sharp(x) \setminus env^\sharp(v) \text{ else } env^\sharp(x)$$

$$\left(env^\sharp, Done\right) \xrightarrow{assume(done())} \left(env^\sharp, \{T\}\right) \qquad \{F\} \neq Done$$

$$\left(env^\sharp, Done\right) \xrightarrow{assume(!done())} \left(env^\sharp, \{F\}\right) \qquad \{T\} \neq Done$$

$$\left(env^\sharp, Done\right) \xrightarrow{return} \left(env^\sharp, Done\right))$$

Fig. 6: Abstract semantics. $\varphi(z) = z' = z \vee z' \in [\![z]\!]_{E_F} \vee [\![z]\!]_{E_L}$

When an abstract instrumented state $\iota^\sharp = (A, \iota)$ represents an instrumented state $((in, out_F, out_L, env), \iota^c)$, A's *Done* component conservatively tracks whether all the input characters in *in* have been read and that the values *env* gives to variables agree with the ones provided by the abstract environment. The instrumentation ι^c of the concrete state should track no less information regarding the information flow of characters from the input string to the output string as does the instrumentation ι. The latter should also consider as bad any bad character in ι^c.

5.2 Abstract Transformers

The abstract transformers are defined by replacing the concrete component in the transition rules of the instrumented semantics with the rules pertaining to abstract states defined in Figure 6 and adapting the rules in Figure 5 to utilize an abstract environment instead of a concrete one as explained below

$$\frac{m^\sharp \xrightarrow{cmd} m^{\sharp\prime} \quad \iota \xrightarrow{cmd}_{m^\sharp} \iota'}{(m^\sharp, \iota) \xrightarrow{cmd} (m^{\sharp\prime}, \iota')}$$

Again, the transition relation of the instrumented part of the state is parameterized with the source abstract state of the transition because it requires access to the abstract envi-

ronment. The required adaptation of Figure 5 is rather direct: where ever an expression of the form $\{env(x)\}$ appears in a rule, we replace it with $env(x)$.

The rules are quite simple; the tricky ones pertain to `assume` statements regarding inequalities, which we now explain.

The abstract transformer of command `assume`(v!=`const`) blocks the execution if the only possible value of v is `const`. Otherwise, it merely records that `const` is not in fact a possible value of v. Note that not only $env^\sharp(v)$ may be adapted, but in case v got its value from the input string, any variable who got its value from the same read operation, i.e., in the same equivalence class as v in either E_F or E_L, may have the set of its possible values refined.

The abstract transformer of command `assume`(v!=`const`) blocks the execution if the only possible value of v is `const`. Otherwise, it merely records that `const` is not in fact a possible value of v. The abstract transformer of command `assume`(v!=y) blocks the execution if the abstract environment associates both variables with the same single character. Otherwise it attempts to refine the set of possible values of one variable if the other one is associated with a singleton set.

Main Theorem The static analysis algorithm computes at every program point an abstract state which over-approximates any instrumented state which can arise at this point for some input string. We denote by $BAD(p)$ the union of the BAD sets at $p()$'s exit points, i.e., right after $p()$ executes a return command. Our main theorem, whose proof follows directly from Lemma 2 and the soundness of the analysis, states that the analyzer computes a set of good characters $p()$.

Theorem 1 *Let $p()$ be an SMP. $\Sigma \setminus BAD(p)$ is a* good *set of characters for $p()$.*

6 Learning Pseudo-Inverse Functions

Our overall goal is that given an SMP p and a desired output string s' to find a string s such that $p(s) = s'$. One natural candidate for s is s' itself. Thus, when trying to learn an inverse we look for an input $s \neq s'$ such that $p(s) = s'$ and hence when generating the training examples, we only use ones where the input is different from the output. Also, if $p()$ is not injective, then it may have many pseudo-inverses, and there is no a priori way to favor any of them. Thus, it suffices to learn an *arbitrary* pseudo-inverse of $p()$.

The learning algorithms chosen to be employed in this paper are the ones we thought handle best the SMPs we have examined. However, they can be easily interchanged with others—our approach, as we said before, is indepedent of the chosen learning algorithm.

6.1 Learning Transducers with OSTIA

Transducers are deterministic finite state machines that are used to translate strings. We explain them using the example transducer depicted in Figure 7. Just like DFAs, transducer read their input strings from the left to the right, character by character, and

```
char* ReplaceSpaces(char *in) {
  char *out = malloc(MAX_STRLEN),
       *s = out, c = 0, skip_next_lf = 0;
  while (*in != 0) {
    c = *in; in++;
    if (skip_next_lf) {
      skip_next_lf = 0;
      if (c == '#') {
        c = *in; in++;
        if (!c) break; }}
    if (*c == '$') {
      skip_next_lf = 1;
      c = '#'; }
    *s = c; s++;}}
  *s = *in;
  return out;}
```

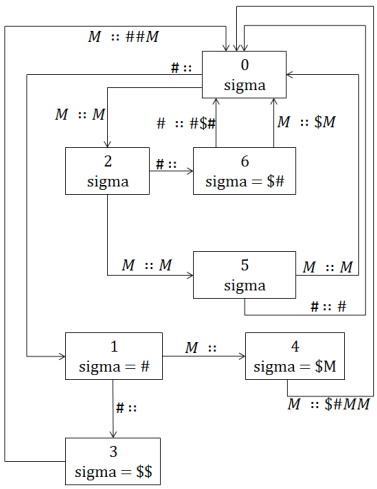

Fig. 7: An SMP written in C and its pseudo-inverse as learned by OSTIA.

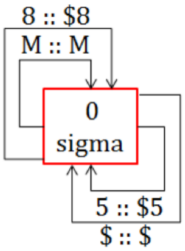

Fig. 8: A transducer implementing the SMP in Figure 2

traverse edges according to a transition function. In addition, as edges are traversed, the transducer prints characters to the output. If the input string *MMMM##* is fed to the transducer, it will go through states $0,2,5,0,2,6,0$, and print the output string *MMMM#$#*. Any states of the transducer can hold inner strings. If some state q is a final state for the transduction, and it holds an inner string *sigma*, then it is appended at the end of the output. For example, the transduction of *MMMM#* equals *MMMM$#*.

OSTIA [14] is a supervised learning algorithm that is capable of learning transducers. Assuming the training set is without noise, like in our case, OSTIA is guaranteed to converge to the real transducer as the size of the training set increases. The SMP from Figure 2 and its inverse are both transducers, and we depict them in Figures 1 and 8, respectively. Every state of the transducer is depicted as a square containing a unique identifier, with state 0 being the initial state, and the string *sigma* which is appended to the output if the transduction end at that state. Transitions between states are depicted

as edges annotated with $\sigma :: s$ denoting the character σ which triggers the transition and the string s appended to the output due to taking the transition.

OSTIA succeeds in learning the exact inverse at the righthand side of the figure. In its essence, OSTIA is an iterative state merging algorithm. At each step the algorithm considers pairs of states as candidates for a possible merge, and if the resulting merged transducer is consistent with the training set, it accepts the merge and proceeds to the next iteration. The transducer in Figure 7 is the pseudo inverse OSTIA learns for the SMP shown in the same figure. The character # in an output string could have originated from either #, $ or $# in the input string. While randomizing our input for the training set, all three possibilities introduce themselves. This is evident in the transducer, as the edges $(5,0),(2,6),(6,0)$ choose a different source for the # character each. Thus neither # nor $ can be good characters.

6.2 Needleman Wunsch Alignment Algorithm

To show the versitility of our approach, we also used the alignment algorithm of Needleman-Wunsch [20] to learn procedure inverses. The algorithm is designed to align input and output strings, where the latter comes from the former by performing a sequence of steps. In each step a character is either deleted, inserted or replaced by another character. Naturally, as the number of steps is smaller, and the input and output are close in terms of edit distance, the results of the alignment are better. Our application uses a random set of inputs $\{s_i\}_{i=1}^n$ just as before, and apply p on each element of the set to end up with a training set $\{(s_i, p(s_i))\}_{i=1}^n$. Note, the order of the training set has changed, as we now want to learn the effect of the *original* SMP p. Each (input,output) pair is then aligned, and three probability tables are accumulated for the original SMP p: (1) A two dimensional table $T_r(p)$ for character replacements, in which $T_r(p)[\$][\#] = 0.45$ means that there is an estimated probability that a $ in the input string will be replaced by a #. (2) A one dimensional table $T_d(p)$, in which $T_d(p)[*] = 0.95$ means there is a probability of 0.95 that $*$ will be deleted from the input string. (3) A one dimensional table $T_i(p)$, in which $T_i(p)[@] = 0.55$,means there is a 0.55 probability of inserting @ *somewhere* in the output. Once these tables have been learned for the original SMP p, they can be used to deduce pseudo inverses p^{-1}: If $T_r(p)[\$][\#] = 0.45$ then clearly $T_r(p^{-1})[\#][\$] = 0.45$. Deducing $T_i(p^{-1})[*]$ based on $T_d(p)[*]$ is a little more subtle, and should also take into account the prevalence of the character $*$ in the input strings of the training set. Note that for more accurate results, $T_i(p^{-1})[*]$ depends on the length of the string y it wishes to invert. Finally, computing $T_d(p^{-1})[@]$ from $T_i(p)[@]$ depends on the prevalence of the character @ in the output strings of training set, the prevalence of @ in y, and the length of y too. It is important to stress out that the resulting pseudo inverse p^{-1} is *not* deterministic, and could return different outputs when invoked multiple times. This can be seen as an advantage, because of $p^{-1}(y)$ failed to find a relevant x, we do not have to perform the learning process again, but simply call $p^{-1}(y)$ again.

7 Implementation and Experimental Evaluation

We have implemented our ideas in a publicly available tool called StrInver. The tool gets as input an SMP $p()$ written in LLVM [18] intermediate representation language,

Procedure	Reduced alphabet	ML. Alg.
DPSTrim()	$\{M,\#\}$	Needleman Wunsch
escapeWS()	$\{\$,5,8,M\}$	OSTIA
ReplaceSpaces()	$\{\$,\#,M\}$	OSTIA
DosNames()	$\{.,\_,M\}$	OSTIA

Table 1: Experimental evaluation of selected SMPs. The table shows the size of the reduced alphabet and the machine learning algorithm we used to model the pseudo inverse.

and a concrete query string y. (In our experiments, we used procedures written in C, compiled using Visual C 2010.) The tools checks whether the procedure falls within the class of restricted SMPs we handle (see Section 3) by expecting it to follow certain syntactic conventions, and if so it looks for a string s such that $s \neq s'$ and $p(s) = s'$. If the learning algorithm failed to find a model that returns a non-identity inverse for the given string s', it is retried with a new randomized training set. The algorithms were trained using a training set comprised of 64-100 examples, with a bias towards choosing shorter strings.

Table 1 summarizes our experimental results. We considered four string manipulating procedures coming from real-life software. `DPSTrim()` removes prefixes and suffixes comprised of character #. It is taken from *DataparkSearch* [1] open source search engine and is used to help parse configuration files. `escapeWS()` is our running example shown in Figure 1 and `ReplaceSpaces()` is shown in Figure 7. Both come from GCC. `DosNames()` is a python library function which replaces all the dots in a file name with underscores, except for the last one.

In our experiments, we randomly chose output strings using uniform distribution and with average length of 32 characters. We applied our technique to invert 100 strings for and each procedure. Table 1 shows the reduced alphabet our analysis discovered and the machine learning algorithm which we used. We ran our experiments in a laptop equipped with an *i*5 2.3*Ghz* CPU with 6GB memory running Windows 7. In all our experiments, it took our analysis to invert each string less than 10 seconds, whereas KLEE [2], a state of the art symbolic executor, failed to invert any string after running for one hour. (KLEE was able to invert short strings containing around 5 characters in a few seconds.) Similarly, a machine learning algorithm trained with randomly selected strings chosen using the full alphabet failed to invert the given output strings. It might be the case that using a larger training set would make the naive machine learning more successful, however, this process might lead to expensive analyses as the space of possible strings grows exponentially with the length of the string.

8 Related Work, Conclusion and Future Work

Automatic inversion of programs was first studied by Dijkstra who manually inverted simple array-manipulating programs [5]. Follow up works looked at inverting (i) sim-

ple programs whose semantics is given as logic programs [26], (ii) tree-traversal programs using relational calculus and deductive methods [3, 29], (iii) array transformers using techniques based on LR-parsing [8, 16] or testing [15], and (iv) bijective string-manipulating procedures [13, 19]. To the best of our knowledge, we are the first to apply machine learning tools to invert programs. We also note that the programs we invert are not necessarily injective.

Recent advances in machine learning lead researchers to explore its capabilities in helping challenging program analysis tasks, e.g., specification inference [25, 28], speed up abstraction refinement [9], invariant generation [7, 22, 27], setting up parameters for parametrized static analyses [24], and infer clustering of variables in partially relational static analyses [12]. In our work, we address a dual question–how can machine learning technique help program analyses. To the best of our knowledge the question has not been widely addressed, with the notable exception of [21] which also argues that a combination of machine learning and program analysis can be a win-win situation.

Another active research area is the use of input/output examples to learn computer programs. Often, this is done in the context of synthesis, where examples guide a search-based synthesis process [11]. For example, in [10], a learning procedure is used to synthesize string manipulating procedures which appears in the context of spreadsheets based on syntactic manipulation. Another attack on this problem was taken in [30], where the procedures where synthesized using database-like lookup operations. In these works, the focus is on designing a language in which programs can be synthesized and an efficient search heuristics. In this work we too focus on string manipulating procedures (SMPs), which are abundant in almost all software packages. However, instead of asking the user for input/output examples, we analyze the code of one procedure and its behavior, as expressed by input-output pairs, to synthesize another procedure.

In [33], the authors suggest to learn the behavior of a procedure by inspecting its code and input-output examples. Their technique applies to a class of procedures which accepts their input character by character, e.g., multi-digit addition. They use recurrent neural network models with long short-term memory to accurately learn a model of the procedure behavior as a sequence-to-sequence transformer [31]. It can be interesting to see if a preliminary phase of program analysis, as we do in this work, can help improve the accuracy of their technique.

String solvers, e.g., [17, 34], can reason about constraints involving operations on strings. For example, HAMPI [17], can reason about constraints expressing membership in regular languages and fixed-size context-free languages. In contrast, we provide a technique based on a combination of machine learning and static analysis that can help invert string manipulating procedures written in a restricted programming language.

Conclusion and Future Work We present a machine learning-based approach for inverting string manipulating procedures (SMPs). To the best of our knowledge, the use of machine learning for program inversion is novel. We make the approach feasible by developing a static analysis which reduces the size of the alphabet of the examples used during training. While the idea of reducing the input domain size is a known idea, e.g., it is common to reduce the bitwidth of integers before performing testing of a compiler optimization, we believe that we are the first to design a static analysis specific for

enabling such a reduction. We evaluated our technique using a small selection of procedures taken from real-life software. We have shown that our technique can compute the inputs required to drive an SMP to produce desired output whereas KLEE—a state of the art symbolic execution engine fails to do so. Our approach does not require that the inverted SMP be bijective. However, our analysis is beneficial when the SMP acts as the identity on a large part of its alphabet, which we refer to as the "good" characters. In the future, we plan to remove this limitation by allowing the SMP to apply invertible transformations to good characters before moving them into the output.

References

1. Http://www.dataparksearch.org/
2. Cadar, C., Dunbar, D., Engler, D.: 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. pp. 209–224. OSDI'08, USENIX Association, Berkeley, CA, USA (2008)
3. Checn, W., Duding, J.T.: Program inversion: More than fun! Science of Computer Programming **15**(1), 1 – 13 (1990)
4. De Moura, L., Bjørner, N.: 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. pp. 337–340. TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (2008)
5. Dijkstra, E.W.: Program inversion. In: Program Construction, International Summer Schoo. pp. 54–57. Springer-Verlag, London, UK, UK (1979), http://dl.acm.org/citation.cfm?id=647639.733360
6. Ganesh, V.: Decision Procedures for Bit-vectors, Arrays and Integers. Ph.D. thesis, Stanford, CA, USA (2007), aAI3281841
7. Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 499–512 (2016)
8. Glück, R., Kawabe, M.: A method for automatic program inversion based on lr(0) parsing. Fundam. Inform. **66**, 367–395 (07 2005)
9. Grigore, R., Yang, H.: Abstraction refinement guided by a learnt probabilistic model. SIGPLAN Not. **51**(1), 485–498 (Jan 2016). https://doi.org/10.1145/2914770.2837663
10. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. SIGPLAN Not. **46**(1), 317–330 (2011)
11. Gulwani, S.: Programming by examples: Applications, algorithms, and ambiguity resolution. In: Proceedings of the 8th International Joint Conference on Automated Reasoning - Volume 9706. pp. 9–14. Springer-Verlag New York, Inc., New York, NY, USA (2016)
12. Heo, K., Oh, H., Yang, H.: Learning a variable-clustering strategy for octagon from labeled data generated by a static analysis. In: Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings. pp. 237–256 (2016)
13. Hu, Q., D'Antoni, L.: Automatic program inversion using symbolic transducers. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 376–389. PLDI 2017, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3062341.3062345, http://doi.acm.org/10.1145/3062341.3062345
14. Jose Oncina, P.G., Vidal, E.: Learning subsequential transducers for pattern recognition interpretation tasks. IEEE Trans. Pattern Anal. Mach. Intell. **15**(5), 448–458 (1993)
15. Kanade, A., Alur, R., Rajamani, S., Ramanlingam, G.: Representation dependence testing using program inversion. In: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 277–286. FSE '10, ACM, New York, NY, USA (2010). https://doi.org/10.1145/1882291.1882332, http://doi.acm.org/10.1145/1882291.1882332
16. Kawabe, M., Glück, R.: The program inverter lrinv and its structure. In: Hermenegildo, M.V., Cabeza, D. (eds.) Practical Aspects of Declarative Languages. pp. 219–234. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)

17. Kiezun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: Hampi: A solver for string constraints. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis. pp. 105–116. ISSTA '09, ACM, New York, NY, USA (2009). https://doi.org/10.1145/1572272.1572286, http://doi.acm.org/10.1145/1572272.1572286
18. Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization. pp. 75–. CGO '04, IEEE Computer Society, Washington, DC, USA (2004)
19. Miltner, A., Fisher, K., Pierce, B.C., Walker, D., Zdancewic, S.: Synthesizing bijective lenses. Proc. ACM Program. Lang. **2**(POPL), 1:1–1:30 (Dec 2017). https://doi.org/10.1145/3158089, http://doi.acm.org/10.1145/3158089
20. Needleman, S.B., Wunsch, C.D.: A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of Molecular Biology **48**(3), 443–453 (1970)
21. Nori, A.V., Rajamani, S.K.: Program Analysis and Machine Learning: A Win-Win Deal. In: Proceedings of the 18th International Static Analysis Symposium. Lecture Notes in Computer Science, vol. 6887, pp. 2–3. Springer International Publishing (2011)
22. Nori, A.V., Sharma, R.: Termination proofs from tests. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. pp. 246–256. ESEC/FSE 2013, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2491411.2491413
23. Octeau, D., Jha, S., Dering, M., McDaniel, P., Bartel, A., Li, L., Klein, J., Le Traon, Y.: Combining static analysis with probabilistic models to enable market-scale android inter-component analysis. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 469–484. POPL '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2837614.2837661
24. Oh, H., Yang, H., Yi, K.: Learning a strategy for adapting a program analysis via bayesian optimisation. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 572–588. OOPSLA 2015, ACM, New York, NY, USA (2015). https://doi.org/10.1145/2814270.2814309
25. Raychev, V., Bielik, P., Vechev, M., Krause, A.: Learning programs from noisy data. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 761–774. POPL '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2837614.2837671
26. Ross, B.J.: Running programs backwards: The logical inversion of imperative computation. Formal Aspects of Computing **9**(3), 331–348 (May 1997)
27. Sankaranarayanan, S., Chaudhuri, S., Ivančić, F., Gupta, A.: Dynamic inference of likely data preconditions over predicates by tree learning. In: Proceedings of the 2008 International Symposium on Software Testing and Analysis. pp. 295–306. ISSTA '08, ACM, New York, NY, USA (2008). https://doi.org/10.1145/1390630.1390666
28. Sankaranarayanan, S., Ivančić, F., Gupta, A.: Mining library specifications using inductive logic programming. In: Proceedings of the 30th International Conference on Software Engineering. pp. 131–140. ICSE '08, ACM, New York, NY, USA (2008). https://doi.org/10.1145/1368088.1368107
29. Schoenmakers, B.: Inorder traversal of a binary heap and its inversion in optimal time and space. In: Bird, R.S., Morgan, C.C., Woodcock, J.C.P. (eds.) Mathematics of Program Construction. pp. 291–301. Springer Berlin Heidelberg, Berlin, Heidelberg (1993)
30. Singh, R., Gulwani, S.: Learning semantic string transformations from examples. Proc. VLDB Endow. **5**(8), 740–751 (Apr 2012). https://doi.org/10.14778/2212351.2212356

31. Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. In: Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada. pp. 3104–3112 (2014)
32. Yi, K., Choi, H., Kim, J., Kim, Y.: An empirical study on classification methods for alarms from a bug-finding static c analyzer. Inf. Process. Lett. **102**(2-3), 118–123 (Apr 2007). https://doi.org/10.1016/j.ipl.2006.11.004
33. Zaremba, W., Sutskever, I.: Learning to execute. CoRR **abs/1410.4615** (2014)
34. Zheng, Y., Ganesh, V., Subramanian, S., Tripp, O., Dolby, J., Zhang, X.: Effective Search-Space Pruning for Solvers of String Equations, Regular Expressions and Length Constraints, pp. 235–254. Springer International Publishing (2015)

A Proof of Lemma 2

Lemma 2. Let $p()$ be an SMP and s, s' strings. If $\hat{p}(s) = (s', BAD)$ then $\Sigma \setminus BAD$ is a *good* set of characters for $p()$ and s.

Proof. Let π be a concrete run of the instrumented $\hat{p}(s)$ on the input string s. A set of characters, *BAD*, is computed during π. Let $\sigma \notin BAD$ be some character in s that was not placed in *BAD*, and suppose without loss of generality that it was read with `x = read-first()`. Immediately after reading it, E_F in the instrumented concrete semantics contains (x,x). Inspecting the transformers, the only way (x,x) can be removed from E_F, is either by marking its content $\rho(x) = \sigma$ as bad, or by performing a `write-first`(x). the former option is infeasible since $\sigma \notin BAD$, and thus σ will be written to the output. Once it is written, it is removed from E_F, and any extra attempt to write, will cause it to be marked as *BAD*, which is not feasible as just mentioned. So σ is copied exactly one time to the output.

Assume now that s contains some two characters σ_1, σ_2, such that σ_1 occurs before σ_2. First, assume that both characters were read from the input, and consider first case where both were read using `read-first()`. The semantics of our programming language ensures that in this case, σ_1 was read before σ_2, as it appears before it in the input. Let us describe all possible scenarios: (1) By the time σ_2 is read, σ_1 has already been removed from E_F. This scenario is possible only if it was written using `write-first`, as other options necessarily mark it *BAD*. (2) when σ_2 is read, σ_1 is still in E_F, so (x,y) will be entered to R_F and then the writing order is enforced. The other cases are proven in a similar manner.

The case where both σ_1 and σ_2 were read using `read-last()` is analogous.

If σ_1 was read `read-first()` and σ_2 using `read-last()`, then they must be written by `write-first()` and `write-last()` respectively, otherwise one of them should have been marked as *BAD*.

Finally, if either σ_1 or σ_2 was not read at all, then it can be removed from s and still have $\hat{p}$ produce the same output. Then the above argument can be repeated, inductively, for the smaller string, which has the same set of good characters as s.