Decision Procedures for String Constraints

Ph.D. Dissertation Proposal
Pieter Hooimeijer
pieter@cs.virginia.edu

April 22, 2010

1 Introduction

Reasoning about string variables is a key aspect in many areas of program analysis [9, 29, 37, 41, 45] and automated testing [16, 17, 18, 27]. Program analyses and transformations that deal with string-manipulating programs, such as test input generation for legacy systems [25, 26], web application bug finding [41], and program repair [42], invariably require a model of string manipulating functions.

Traditionally, both static and dynamic analyses have relied on their own built-in models to reason about constraints on string variables, just as early analyses relied on built-in conservative reasoning about aliasing. The current situation is suboptimal for two reasons: first, it forces researchers to re-invent the wheel for each new tool; and secondly, it inhibits the independent improvement of algorithms for reasoning about strings.

External constraint solving tools have long been available for other domains, such as satisfiability modulo theories (SMT) [11, 12, 32] and boolean satisfiability (SAT) [13, 31, 44]. These tools are used, for example, to rule out infeasible program paths or, given a program path, to generate testcases that exercise that path. Recent work in string analysis has focused on providing similar external decision procedures for string constraints [2, 21, 24, 39, 40, 46]. Thus far, this work has focused on individual features, such as support for symbolic integer constraints [46], support for bounded context-free grammars [2, 24], and embedding into an existing SMT solver [11, 39].

These recent approaches are motivated by the fact that an important subclass of software defects is caused by the improper handling of structured text such as HTML, XML, and SQL [37, 38]. Two compelling examples of this type of defect are SQL injection and cross-site scripting vulnerabilities. These vulnerabilities are common; together they accounted for 35.5% of reported security vulnerabilities in 2006 [20]. A November 2009 study found that 64% of the 1,364 surveyed websites had at least one serious vulnerability [36], where *serious* means "Exploitation could lead to serious and direct business impact." Security vulnerabilities are also costly; in a 2008 FBI survey of over 500 large firms, the average reported annual cost of computer security defects alone was $289,000 [34, p.16].

For this dissertation, we propose work on decision procedures for string constraints. Ultimately, our goal is to develop a decision procedure that simultaneously satisfies these criteria:

- **Scalability**. Good performance is crucial if string decision procedures are to see significant use. We base this observation on the fact that, in a similar way, boolean SAT solvers (e.g., [31,

13]) did not find widespread use until significant engineering effort was put into making them perform well. In Section 4.1, we demonstrate that existing string decision procedures leave room for improvement when it comes to scalability.

- **Expressive Utility**. String decision procedures cannot efficiently support all common string operations. In Section 4.2, we demonstrate that existing approaches do not cover a large class of string operations that use integers to represent positions in strings. How to include these operations in an efficient constraint solver is an open problem.

- **Correctness**. Decision procedures are particularly well-suited for formal correctness arguments because their correctness conditions (soundness and completeness) are often easy to describe formally in a succinct way. In addition, if decision procedures are to be used for conservative program analyses, then it is crucial that they be certifiably correct.

Existing work in this area generally trades off performance for expressive utility. For example, Rex [40] is built on top of Z3, a general purpose SMT solver, and consequently gets support for integer arithmetic queries "for free." This functionality carries a massive cost, however; Rex is several orders of magnitude slower than the Hampi tool [24], which is more performance-oriented but less featureful. Both Rex and Hampi rely on relatively complex underlying constraint solvers (Z3 [11] and STP [15], respectively), and their correctness relies crucially on the underlying code base and the correct use of its interface.

Our approach is based on two insights: (1) string constraint solving can be cast as an explicit search problem, and (2) we can instantiate the search space lazily through incremental refinement. These insights lead to substantial performance gains relative to eager approaches; our preliminary results show our prototype to be 100× faster on average over a previously published benchmark. We hypothesize that this algorithm can be extended to efficiently support a large subset of commonly used string functions, including those that use integers to represent string positions and lengths. Finally, we believe that such an algorithm can be constructed based on a relatively small trusted code base relative to to existing approaches, leading to the following thesis statement:

> It is possible to construct a practical algorithm that decides the satisfiability of constraints that cover both string and integer index operations, scales up to real-world program analysis problems, and admits a machine-checkable proof of correctness.

The expected main contributions of the proposed dissertation are as follows:

1. A certified automaton-based algorithm, concat_intersect, and associated tool (*Decision Procedure for Regular Language Equations (DPRLE)* [21]) for solving string constraints (Section 5.1).

2. A lazy search-based algorithm for solving constraints and the evaluation of its performance characteristics relative to DPRLE and Hampi (Section 5.2).

3. The extension of the feature set of (2) to support common string operations, guided by a study of string function usage in real-world code, and its evaluation relative to existing tools for both string constraints and integer arithmetic (Section 5.3)

4. Time permitting, an investigation of proof strategies for the full correctness of an algorithm similar to (3) (Section 5.4).

This proposal is structured at follows. In Section 2 we provide an indicative real-world piece of code that makes use of low-level string functions. Section 3 presents a basic constraint language that captures the functionality offered by existing tools. Section 4 presents preliminary results that motivate our use of a lazy search strategy (scalability; Section 4.1), and our choice of features (expressive utility; Section 4.2). Section 5 provides a high-level overview of the proposed dissertation, and Section 6 discusses the benchmarks we will use to evaluate our approach. Section 7 provides a concrete time line for this dissertation. We conclude in Section 8.

2 Motivating Example

In this section we present a code fragment taken from version 2.6.0 of `wu-ftpd`, an file transfer server with a known format string vulnerability. This code demonstrates the need for analyses that are able to reason about the run-time values of string variables. More concretely, it demonstrates an example where both test input creation and program repair require efficient and correct reasoning about both strings and integers.

Figure 1 shows a fragment of code for handling `SITE EXEC` commands from `wu-ftpd`. The `SITE EXEC` portion of the file transfer protocol allows remote users to execute certain commands on the local server. The **`cmd`** string holds untrusted data provided by such a remote user; an example benign value is **`"/usr/bin/ls -l *.c"`**.

The variable **`_PATH_EXECPATH`** points to a directory containing executable files that remote users are allowed to invoke (e.g., **`"/home/ftp/bin"`**). To prevent the remote user from invoking other executables via pathname trickery (e.g., **`cmd == "../../../bin/dangerous"`**), lines 5–12 sanitize the command string by skipping past all slash-delimited path elements. However, skipping past all slashes does not have the desired effect: **`"/bin/echo '10/5=2'"`** should become **`"/echo '10/5=2'"`** and not **`"5=2'"`**; slashes should only be removed from the command, not from the arguments. The **`strchr`** invocation on line 4 is used to check if any spaces are present (line 6). If so, a more complicated version of the slash-skipping logic is used (lines 10–11) that only advances **`cmd`** past slashes before the first space. Lines 15–18 build the command that will be executed (e.g., completing the transformation from **`"/usr/bin/ls -l *.c"`** to **`"/home/ftp/bin/ls -l *.c"`**) by using **`sprintf`** to concatenate the trusted directory, a slash, and the suffix of the user command. The check on line 16 prevents a buffer overrun on the local stack-allocated variable **`buf`** by explicitly adding together the two string lengths, one byte for the slash, and one byte for C's null termination, and comparing the result against the size of **`buf`**.

This code is indicative of real-world string processing: it is complicated, it accomplishes many goals simultaneously, and it involves control flow and imperative updates based on the interactions between strings and integers. Consider the buffer overrun check on line 16: it is much harder to verify than a **`snprintf(buf,sizeof(buf),...)`** invocation, but both are correct. In addition, note that the bounds checking is done *after* **`cmd`** has been advanced past any slashes. Thus it is possible for the user to provide input that is initially too long for **`buf`** but that fits safely after the sanitization. The vast majority of static buffer overrun analyses would produce a false positive on this code.

More tellingly, while the code correctly avoids buffers overruns and implements its path-based security policy, it is vulnerable to a format string attack [6]. Since the user's command is passed as the format string to **`fprintf`** (line 20), if it contains sequences such as **`%d`** or **`%s`** they will be

```
1  void site_exec(char *cmd)
2  {
3    char buf[MAXPATHLEN], *slash, *t;
4    char *sp = (char *) strchr(cmd, ' ');
5    /* sanitize the command-string */
6    if (sp == 0) {
7      while ((slash = strchr(cmd, '/')) != 0)
8        cmd = slash + 1;
9    } else {
10     while (sp && (slash = (char *) strchr(cmd, '/')) && (slash < sp))
11       cmd = slash + 1;
12   }
13   for (t = cmd; *t && !isspace(*t); t++)
14     if (isupper(*t)) *t = tolower(*t);
15   /* build the command */
16   if (strlen(_PATH_EXECPATH) + strlen(cmd) + 2 > sizeof(buf))
17     return;
18   sprintf(buf, "%s/%s", _PATH_EXECPATH, cmd);
19   /* ... execute buf, store results ... */
20   fprintf(remote_socket, cmd); /* tell user final command */
21   /* ... copy results back to user via remote_socket ... */
22 }
```

Figure 1: Source code using hand-written sanitization and checks to avoid a buffer overrun (successfully, line 18) and a format string vulnerability (unsuccessfully, line 20), and enforce path-related policies (successfully). The string `cmd` contains untrusted data provided by a remote user. Functions that require explicit reasoning about both integers and strings are highlighted (e.g., `strchr`).

interpreted by `printf`'s formatting logic. This typically results in random output, but careful use of the uncommon `%n` directive, which instructs `printf` to store the number of characters written so far through an integer pointer on the stack, can allow an adversary to take control of the system. An exploit for just such an attack against exactly this code was made publicly available [43].

Locating this defect and constructing a testcase can both be viewed as solving a system of constraints induced by the program. For example, locating the attack is equivalent to asking: "is it possible for `cmd` to have the properties of a format string attack (e.g., be a string of a certain minimal length, containing both shell code payload and the `'%n'` character) at the end of any path that reaches line 20?" At a very high level, this becomes a question of language emptiness after intersection: "is the result of intersecting the set of format string attacks with the set of strings that contain no slashes before the first space empty?" The testcase generation problem uses the same set of constraints, but asks for members of the set, rather than a boolean indication of the set's emptiness.

While there are a number of string analyses that can handle constraints over systems of string variables, there are no scalable analyses that can handle mixed integer and string constraints. Solving constraints about the program in Figure 1 requires not just knowing that `strchr` returns an integer indicating the presence or absence of a special substring, but requires understanding that the integer is an *index* into to the location of that substring, and thus that the subsequent pointer arithmetic is advancing past it. We propose to create and evaluate such an analysis.

$Constraint$	::=	$StringExpr \in RegExpr$	inclusion	$RegExpr$	::=	$ConstVal$	string literal
	|	$StringExpr \notin RegExpr$	non-inclusion		|	$RegExpr + RegExpr$	language union
$StringExpr$	::=	Var	string variable		|	$RegExpr\ RegExpr$	language concat
	|	$StringExpr \circ Var$	concat		|	$RegExpr^\star$	Kleene star

Figure 2: String inclusion constraints for regular sets. A constraint system is a set of constraints over a shared set of string variables; a satisfying assignment maps each string variable to a value so that all constraints are simultaneously satisfied. $ConstVal$ represents a string literal; Var represents an element in a finite set of shared string variables.

3 Definitions

In this section, we define a set of string constraints similar to those presented by Kiezun *et al.* [24]. This definition is representative of the current state of the art for string decision procedures. In this work, we propose decision procedures for extensions to this basic set of constraints.

The set of well-formed string constraints is defined by the grammar in Figure 2. A constraint system S is a set of constraints of the form $S = \{C_1, \ldots, C_n\}$, where each $C_i \in S$ is derivable from $Constraint$ in Figure 2. Var denotes a finite set of string variables $\{v_1, \ldots, v_m\}$. $ConstVal$ denotes the set of string literals. We describe inclusion and non-inclusion constraints symmetrically when possible, using $\diamond$ to represent either relation (i.e., $\diamond \in \{\in, \notin\}$).

For a given constraint system S over variables $\{v_1, \ldots, v_m\}$, we write $A = [v_1 \leftarrow x_1, \ldots, v_m \leftarrow x_m]$ for the *assignment* that maps values $x_1, \ldots, x_m$ to variables $v_1, \ldots, v_m$, respectively. We define $[\![v_i]\!]_A$ to be the value of v_i under assignment A; for a $StringExpr$ E, $[\![E \circ v_i]\!]_A = [\![E]\!]_A \circ [\![v_i]\!]_A$. For a $RegExpr$ R, $[\![R]\!]$ denotes the set of strings in the language $L(R)$, following the usual interpretation of regular expressions. When convenient, we equate a regular expression literal like $\texttt{ab}^\star$ with its language. We refer to the negation of a language using a bar (e.g., $\overline{\texttt{ab}^\star} = \{w \mid w \notin \texttt{ab}^\star\}$).

An assignment A for a system S over variables $\{v_1, \ldots, v_m\}$ is *satisfying* iff for each constraint $C_i = E \diamond R$ in the system S, it holds that $[\![E]\!]_A \diamond [\![R]\!]$. We call constraint system S *satisfiable* if there exists at least one satisfying assignment; alternatively we will refer to such a system as a *yes–instance*. A system for which no satisfying assignment exists is *unsatisfiable* and a *no–instance*.

A *decision procedure* D for string constraints is an algorithm that, given a constraint system S, returns either $D(S) = \mathsf{Sat}(A)$ iff a satisfying assignment exists (where A is such an assignment), or $D(S) = \mathsf{Unsat}$ iff no satisfying assignment exists. More explicitly, such a decision procedure must be *sound*:

$$\forall S, D(S) = \mathsf{Sat}(A) \Rightarrow \forall (E \diamond R) \in S, [\![E]\!]_A \diamond [\![R]\!]$$

and *complete*:

$$\forall S, \mathsf{satisfiable}(S) \Rightarrow D(S) \neq \mathsf{Unsat}.$$

In addition, D must terminate on all well-formed inputs.

We discuss several decision procedures that require a length bound for the strings they output. In this case the soundness argument remains the same, but instead of completeness we prove *bounded completeness*:

$$\forall S, \forall k \geq 0, \mathsf{satisfiable}(S, k) \Rightarrow D(S) \neq \mathsf{Unsat}$$

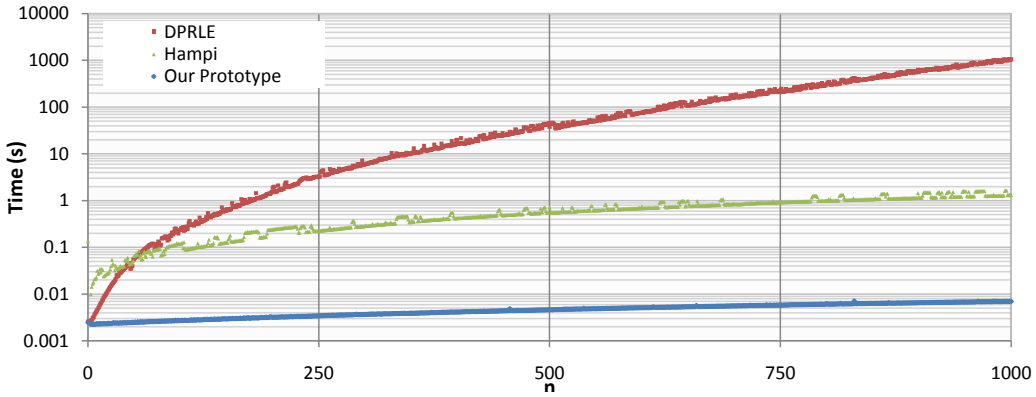

Figure 3: String generation times (log scale) for the intersection of the regular languages `[a-c]*a[a-c]{n+1}` and `[a-c]*b[a-c]{n}`, for n between 1 and 1000 inclusive.

where k is an integer length bound, and satisfiable(S, k) denotes that there exists at least one satisfying assignment for S such that each assigned string value has length less than or equal to k.

4 Preliminary Results

This section describes preliminary results that underscore the need for string decision procedures that can efficiently model a variety of programming idioms related to strings. Section 4.1 demonstrates that a lazy search strategy may yield significantly better performance than existing eager approaches. Section 4.2 presents a study of a large body of PHP code; it suggests that an important class of commonly used string functions are not modeled by existing string decision procedures

4.1 On the Scalability of Existing Approaches

We hypothesize that existing solvers are substantially less scalable than they could be. To test this, we developed a prototype based on our existing DPRLE tool [21]; we will refer to this algorithm and its implementation as *our prototype*. Given two regular expressions, the tool constructs two finite state automata. The conversion algorithm, due to Ilie and Yu [22], provably generates small automata with few redundant states and transitions. We then generate strings from these two automata. We optimized the algorithm for yes-instances — rather than constructing the intersection first and then searching it for strings, the new algorithm interleaves the intersection and string finding steps. The full search space is only constructed for certain no-instances; for yes-instances, the search terminates as soon as a string is found.

To test this hypothesis, we reproduce and extend an experiment used to evaluate the scaling behavior of Rex [40]. We compare the performance of DPRLE, Hampi, and our prototype. The task is as follows. For some length n, given the regular expressions

`[a-c]*a[a-c]{n+1}` and `[a-c]*b[a-c]{n}`

find a string that is in both sets. For example, for $n = 2$, we need a string that matches both `[a-c]*a[a-c][a-c][a-c]` and `[a-c]*b[a-c][a-c]`; one correct answer string is `abcc`. Note

that, for any n, the result string must have length $n + 2$. For Hampi, we specify this length bound explicitly; DPRLE and our prototype do not require a length bound.

For each n, we run the three tools, measuring the time it takes each tool to generate a single string that matches both regular expressions. Figure 3 shows our results. Our prototype is, on average, $118\times$ faster than Hampi; the speedup ranges from $4.4\times$ to $239\times$. DPRLE outperforms Hampi up to $n = 55$, but exhibits considerably poorer scaling behavior than both other tools. By comparison, the published Rex results [40] for $n = 1000$ show that tool taking approximately 140 seconds, or approximately $100\times$ longer than Hampi, and $20,000\times$ longer than our prototype on similar hardware. An informal review of the results shows that our prototype generates only a fraction of the NFA states; for $n = 1000$, DPRLE generates $1,004,011$ states, while our prototype generates just $1,010$ (or just 7 more than the length of the discovered path). These results suggest that lazy constraint solving can save large amounts of work relative to eager approaches.

4.2 A Survey of String Operation Usage

Decision procedures are only useful if their input formulas can encode interesting problems. For example, decision procedures for bit vectors typically include support for bitwise operations and circular shifts [15] on fixed-length arrays of bits. For strings, there is little consensus on which operations to support. Not all operations can be readily implemented in decision procedures; informally, the operation must be reversible in an efficient way. String operations are, in general, both difficult and computationally expensive to reverse. Because of this, we wish to narrow down our effort to a subset of functions that would allow us to reason about a large class of programs.

We consider the top 100 PHP projects listed on the SourceForge source code repository as of 12 December 2009. SourceForge project rankings are based on several statistics, including the number of times the project was downloaded over a period of time. Of these 100 projects, 88 provided readily-accessible source code in the repository at the time of download; the 12 remaining projects were removed from consideration. The final set includes popular projects like phpMyAdmin, a database administration application, and phpBB, a forum application.

The PHP manual lists 113 distinct string functions [28]. We categorized these into five classes:

- **Index.** These operations return an integer or take at least one integer parameter that represents a *position* in or *length* of a string. For example, the function `substr(w,s,l)` returns the substring of string `w` that starts at integer position `s` and has length `l`. This category contains 18 distinct functions.

- **Regular Expression.** These functions have at least one string parameter that denotes a regular expression. PHP broadly supports two classes of regular expressions: a set of POSIX standard functions and a separate Perl Compatible (PCRE) set of functions. This category contains 16 distinct functions.

- **Character.** This category consists of `chr`, which returns the single character associated with an integer, and its inverse, `ord`.

- **Formatting.** These functions, like `printf`, take at least one parameter that represents a format string. This category contains 7 distinct functions.

- **Other.** The majority of the remaining functions are basic re-encoding functions like `trim`, which removes whitespace from its parameter. This category contains 68 distinct functions.

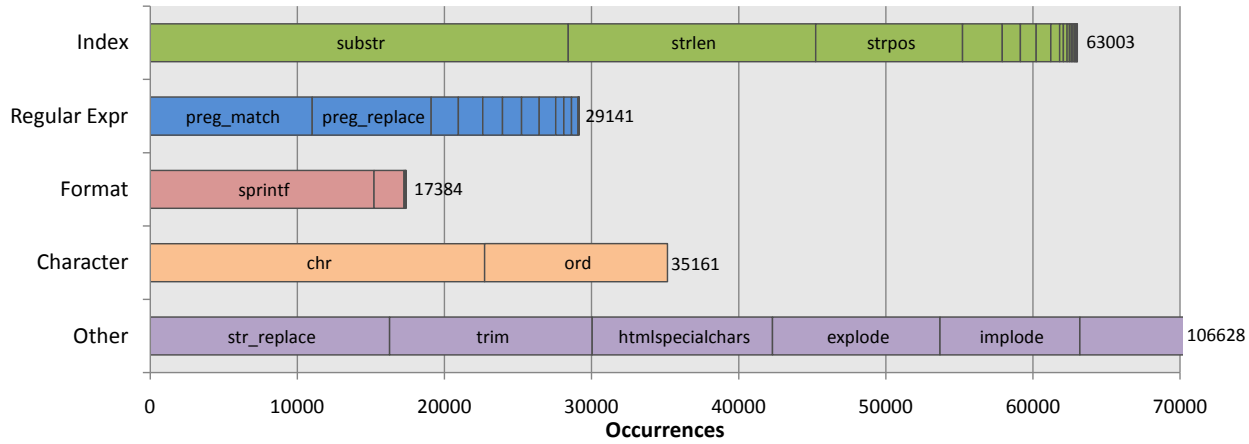

Figure 4: Survey of string function usage (by static count) in the top 88 open source PHP projects, covering 9.6 million lines of code. *Index* functions are used more than twice as frequently as *Regular Expr* functions. Current string decision procedures support the *Regular Expr* category but not the *Index* category; this proposal covers both.

Finally, we disregard the two functions `print` and `echo`, both of which simply output their parameter to the screen and are thus not interesting for our purposes.

We conducted the survey by searching all files with the file extension `.php` in every project. The data set consisted of $40,802$ files containing $9,640,851$ nonempty, non-comment lines of code. We searched each source file for instances of the 111 string manipulating functions of interest. We found $251,317$ such instances. Figure 4 breaks down the results by category; data points with more than $5,000$ occurrences are labeled with the corresponding string function. Each horizontal bar is labeled on the right with the number of total occurrences.

The results show that *Index* functions are used more than twice as frequently as *Regular Expression* functions. To the best of our knowledge, no existing string decision procedure currently supports the functions in *Index* natively. This experiment suggests that, without support for this category of functions, string decision procedures cannot directly model a significant body of existing code. Finally, we observe that *Format* functions are relatively common as well. These functions are of interest because they are susceptible to attacks such as *format string vulnerabilities*. Although this type of vulnerability is typically associated with lower-level languages like C, scripting languages like PHP, Perl, Python, and Ruby can each exhibit similar security flaws [4].

In a smaller study of real-world JavaScript code, Saxena *et al.* find that 83% of string function usage involved an *Index* operation, while regular expression operations were relatively infrequent [35]. Taken together, the results reported here and those reported by Saxena *et al.* motivate our decision to investigate a decision procedure that handles both *Index* and *Regular Expr* operations.

5 Proposed Research

Our goal is to develop scalable, correct and expressive string decision procedures that are suitable for use in program analysis. The motivating example of Section 2 illustrates the need for automatic reasoning about string operations. In Section 4.2, we showed that existing string decision procedures do not support a large class of commonly-used string operations; our motivating example

includes a subset of these operations.

We now outline our research plan for several novel decision procedures. This plan is informed by our previous work on string decision procedures [21, 24], as well as the preliminary results presented in Section 4. Our ultimate goal is to develop a freely available string decision procedure that simultaneously satisfies the following high-level requirements:

- **Scalability**. The decision procedure must scale along two dimensions. The first is the size of the input equations; this is typically measured by the number of variables in the equation, and the size of the regular expressions and/or context-free grammars. The second dimension is the size of the requested output string.
- **Expressive Utility**. The decision procedure must be able to solve equations that map to real and interesting program analysis problems. The tool should be able to solve equations over multiple variables, and the equations should be able to capture common program operations such as string concatenation, equality checks, regular expression checks, substring indexing and substring search.
- **Correctness**. The output of a decision procedure must be trusted since it will be used as a black box by other analyses and transformations. For example, if a decision procedure for strings is used to guide the automated program repair of a security error, developers must have guarantees that the string reasoning is correct. Consequently, we believe it is crucial that the tool be provably correct.

We provide our strategy for attaining these goals in the following subsections.

5.1 A Decision Procedure for Subset Constraints over Regular Languages

The DPRLE tool represents an initial approach that makes heavy use of automata operations. This choice has two important consequences: (1) the core algorithm (concat_intersect) is relatively easy to verify, and (2) the implementation is relatively inefficient in practice. DPRLE is designed for use by traditional string analyses (e.g., [9]) that use automata internally, so the implementation interface is based around automata descriptions rather than regular expressions.

Our presentation of DPRLE includes a correctness proof for the core algorithm that underlies the DPRLE implementation. For this proof, we formalize the correctness conditions of the core algorithm in the calculus of inductive constructions [10]. The proof is from first principles and includes a fully functional re-implementation of the core constraint solving algorithm[1]. This implementation is machine-verifiable—any prospective user can run a proof verifier like Coq [10], and the verifier will automatically establish that the implementation has the advertised formal correctness properties.

A major drawback of this certification approach is that it requires significant implementation effort; for other domains (such as compilers) there is orthogonal work on proof engineering that attempts to address these issues [7, 8]. In particular, any changes to the core algorithm would require corresponding changes to the proof structure. To the best of our knowledge, Minamide's [30] is the only other work to provide a machine-verifiable correctness proof for a string decision procedure. Our experience certifying concat_intersect informs another proposed research thrust on proof engineering strategies for a string decision procedure (Section 5.4).

[1]The proof is available along with the tool at `http://www.cs.virginia.edu/~ph4u/dprle/index.php`

5.2 Solving String Constraints Lazily

Existing string analysis tools tend to rely on solvers for bitvectors [24], boolean SAT [2], or combinations of theories [40]. For these tools, solving a constraint is a three-stage process: (1) the input string constraint is converted to an input for the underlying solver; (2) the underlying solver is called on the converted input; and (3) the results are converted back to a solution for the string constraint system. In an informal study of the Wassermann and Su dataset [24, 41], we found that the main scalability bottleneck of the Hampi tool is, in fact, the first encoding step (and notably *not* the subsequent search).

Approach Our insight is that we can avoid an eager encoding of the string constraints by instantiating the search space lazily. In Section 4.1, we presented promising initial results showing that a lazy search approach can significantly out-perform existing string decision procedures. We propose research on a lazy automaton-based search algorithm that can solve general string constraints systems of the form presented in Section 3. The key challenge for this approach is finding an appropriate separation of the search space description and its instantiation.

The goal for this approach is *not* to rely on underlying decision procedures like SAT. At a high level, we propose two orthogonal approaches to making the tool scale:

- **Search space reduction** explicitly attempts to shrink the total search space that must be traversed in the worst case (e.g., some no-instances). The "follow automata" construction by Ilie and Yu [22], which we use to convert regular expressions to automata, is one such technique.

- **Search heuristics** change the order in which the search space is traversed. The goal is to reduce the average-case search time.

In these approaches we will draw on previous work on dataflow analysis and model checking [1, 19], which face similar state space traversal problems, for inspiration.

At a high level, our algorithm proceeds by iteratively restricting occurrences of variables in the constraint system, using a backtracking search. For clarity, we will distinguish between *restrictions* on variables imposed by the algorithm and *constraints* in the input constraint system. Our search starts by considering all variables to be unrestricted. We then iteratively pick one of the variables to restrict; doing this typically imposes further restrictions on other variables as well.

The order in which we apply restrictions to variables does not affect the eventual outcome of the algorithm (i.e., "Satisfiable" or "Unsatisfiable"), but it may affect how quickly we find the answer. Restrictions are expressed in terms of finite state automata (NFAs); for example, for a constraint $v_1 \circ v_2 \in R_1$, we apply a restriction: "the v_1 occurrence must begin with the start state of the NFA for R_1." Adding restrictions typically involves performing one or more partial NFA intersections.

During the search, if we find that we have over-restricted one of the variables, then we backtrack and attempt a different way to satisfy the same restrictions. At the end of the search, there are two possible scenarios:

- At the end of a successful search, each occurrence of a variable in the original constraint system will be mapped to an NFA path; all paths for a distinct variable will have at least one string in common. We return "Satisfiable" and provide one string for each variable.

- At the end of an unsuccessful search, we have searched all possible NFA path assignments for at least one variable, finding no internally consistent mapping for at least one of those variables. There is no need to explore the rest of the state space, since adding constraints cannot create new solutions. We return "Unsatisfiable."

The algorithm is best described by the inductive invariants that hold for the (implicitly constructed) search tree. Intuitively, each step away from the root must add additional restrictions to variable occurrences. This process is monotonic; once added, a restriction remains in effect for all the vertex' children in the search tree. We backtrack if we find that a variable is over-restricted (i.e., there is no string assignment that could simultaneously satisfy all restrictions imposed by the ancestors in the tree). Repeated backtracking visits to the same vertex will result in a systematic search for ways to apply further restrictions to that vertex' variable-to-work-on. If we are forced to backtrack while considering the root node, then there does not exist a valid string variable assignment, and we report "Unsatisfiable."

Compared to both Hampi and DPRLE, the main benefit of this approach is that, in the optimal case, if we assume a linear-time NFA construction method, then search performance is linear in the length of the output ($\Theta(n \cdot d)$, where d is the average NFA degree and n is the length). Hampi eagerly encodes all regular expressions, which is at best linear, but only for regular expressions that do not contain any positional shifts. DPRLE, finally, would eagerly compute the intersection automaton of the input regular expressions, which is at best linear in the size of the input automata (but only if the automata are identical). While these informal lower bounds are not directly comparable, in practice our prototype implementation is two orders of magnitude faster than Hampi, which in turn, is several orders of magnitude faster than DPRLE.

5.3 Combining String Constraints and Integer Index Operations

We define an *integer index operation* to be a string function that returns an integer or uses at least one integer parameter that represents a *position* in or the *length* of a string. For example, the function `substr(w,s,l)` returns the substring of string `w` that starts at integer position `s` and has length `l`. In Section 4.2, we presented a survey of 9.6 million lines of open source PHP code. The results show that integer index operations occur twice as frequently as the built-in regular expression functions. In Section 2, we show a common programmer idiom for manipulating strings: a loop construction that performs complex path-dependent checks as it traverses the characters of a string.

Our goal is to support regular expression operations (as in previous work [21, 24, 40]), but also to include efficient symbolic constraint solving for integer index functions (Section 4.2). There has been limited work investigating the inclusion of index constraints as part of string decision procedures [3, 23]. This work has focused on the computational complexity and decidability of these operations, but has not lead to a practical algorithm. We believe these operations can be handled integrally, without resorting to an external solver for integer constraints. We will investigate the use of automata-based methods for solving Presburger arithmetic (e.g. [14]).

Approach We hypothesize that a lazy automaton-based search algorithm for string constraints can be efficiently extended to support common indexing operations. We will draw on existing work on Presburger arithmetic using automata [14] and on binary length automata [46]. The

primary challenge is that the existing work does not immediately enable solving over integer index operations.

Our approach will build on the search-based algorithm developed in the previous section. We propose a tightly-integrated incremental algorithm that solves constraints containing both integers and strings. By contrast, a naïve approach might simply alternate between two dedicated algorithms (one for string constraints and one for integer constraints). While conceptually simpler than a tightly-coupled approach, we anticipate (and will evaluate empirically) that the integrated approach will yield better performance.

5.4 Optional: Proof Strategies for String Constraint Solving Tools

Approach Time permitting, we propose two methods for certifying our proposed tools and algorithms. Firstly, we intend to provide a core correctness proof similar in spirit to the DPRLE proof [21]. The proof will cover all of the core steps of the solving algorithm by providing a simplified re-implementation from first principles. The proof will be machine-verifiable. Providing such a proof is a nontrivial task, and we believe building on our previous work will yield a significant direct benefit, as well as a number of indirect ones. It has been our experience that formally proving an algorithm correct in such a system forces the designers to think carefully about, and document, any and all assumptions involved—information that is of paramount importance to later users.

Secondly, we plan a separate effort to make our decision procedure *self-certifying*. This idea is of theoretical interest, and builds on *translation validation* [33]. The key idea is that, in addition to certifying our *algorithm* by providing the correctness proof, we also wish to certify our full *implementation*. Providing a full correctness proof for any nontrivial amount of code is not feasible using today's tools. Instead, we certify individual executions of the decision procedure, by having it optionally emit a full *transcript* for each execution. The transcript will allow a separate verifier tool to assert that the decision procedure operated correctly during that particular execution. The verifier, in turn, is simple enough be furnished with a full correctness proof [5, 32]. Thus every particular run and output of the implementation can be verified before use. We note that, for yes-instances, the satisfying assignment is generally sufficient for validation of the output. For no-instances, however, it is not immediately clear what information is sufficient to validate the answer.

6 Proposed Experiments and Evaluation

In this section, we outline our proposed experimental methodology for each research thrust. Individual parts of the evaluation plan may apply to more than one research thrust. We will evaluate the proposed string decision procedures with respect to our stated goals of *scalability*, *expressive utility*, and *correctness*. The ultimate goal is for our tools to be useful within a wide variety of program analyses.

6.1 A Decision Procedure for Subset Constraints over Regular Languages

This exploratory project emphasizes expressive utility and correctness. We will evaluate whether the DPRLE algorithm is practical for solving a real-world program analysis problem, namely, the

generation of SQL injection attacks based on the output of a static bug finder by Wassermann and Su [41]. We will measure the size of the generated constraint systems for this problem, the time our implementation takes to generate inputs, and we will conduct informal checks to see if the generated attack vectors are accurate. We will consider this approach successful if the total running time does not exceed 10 minutes per generated input. We will provide a machine-checkable proof and include a description of the proof obligations as part of the exposition.

6.2 Solving String Constraints Lazily

This research thrust emphasizes scalability, and its evaluation will rely largely on a within-domain performance comparison with existing tools. We will do a direct comparison between Hampi [24], DPRLE [21], and our new decision procedure. We do not anticipate a comparison with Rex because that tool is not publicly available. It should be noted that the features between these three tools may not overlap perfectly; if so we will choose reasonable common feature subsets and do pairwise comparisons. We will perform separate measurements for yes-instances, for which we will measure the time it takes each tool to generate the first satisfying string for all variables in the constraint system. For no-instances, we will measure the time it takes to report the "Unsatisfiable" answer.

We will run this experiment on several datasets: the Rex dataset [40], a set difference task that consists of 100 regular expression pairs (90 yes-instances, 10 no-instances); the CFG Analyzer dataset, which consists of a large number of variously sized context-free grammars; and the "long strings" benchmark shown in Section 4.1. We hypothesize that our tool will outperform the others by several orders of magnitude, as suggested by the results shown in Figure 3. In the presentation of our data, we will consider two separate dimensions of scalability: input constraint system size and requested output string length. We will consider our tool successful if it is at least one order of magnitude faster than the existing tools on the majority of the testcases.

6.3 Combining String Constraints and Integer Index Operations

We wish to establish that, where features overlap, the performance of this approach is competitive relative to that of existing approaches. We will conduct a within-domain comparison similar to the one proposed in the previous subsection.

The evaluation for this project will emphasize the expressive utility argument. More concretely, we wish to verify that our decision procedure can solve an interesting set of problems. In this case, comparison with previous tools is inadequate, since existing tools do not support both string operations and integer index operations. For this experiment, we will construct a new dataset based on the survey presented in Section 4.2. This dataset will consist of fragments of PHP code that contain at least two data-dependent index functions (for example, code that uses `strchr` to find an index followed by `substr` to extract a substring at that index). We will then execute this code (in context) to generate specific string and integer values and pick random variables to solve for.

We will evaluate our tool on this dataset by measuring how many of these constraint systems our tool is able to solve without modification. We will consider our approach successful if the tool handles more than 80% of the constraints drawn randomly from this new dataset within a 10-second timeout per testcase for reasonably-sized strings. Finally, time permitting, we will seek

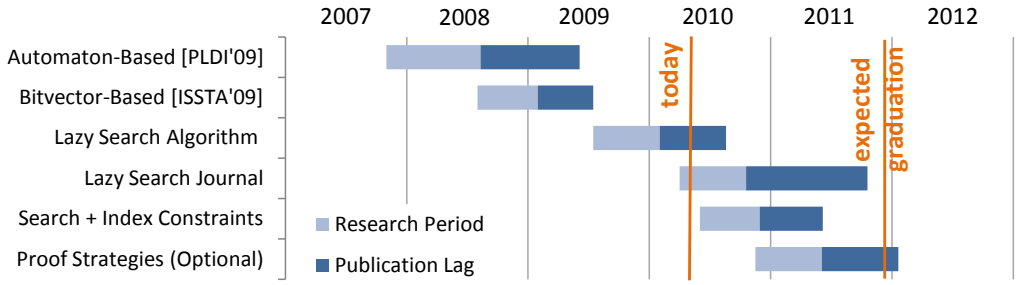

Figure 5: Proposed work schedule. The *Lazy Search Algorithm* paper is currently under submission.

out collaboration in a client application domain such as security or testcase generation.

6.4 Optional: Proof Strategies for String Constraint Solving Tools

Our correctness criterion does not warrant empirical evaluation directly. We will make the correctness argument an integral part of the presentation of our decision procedures. If we choose to pursue the *Proof Strategies* thrust, we will render a machine-checkable proof for at least one of the decision procedures. Our goal is to provide a detailed, from-first-principles correctness proof; a similar proof in our previous work was well-received [21]. If we choose to pursue the *Proof Strategies* thrust on self-certification, we will measure the runtime overhead associated with generating and validating transcripts.

7 Schedule

Figure 5 describes the proposed schedule for this dissertation and related projects. It consists of five related projects:

1. An automaton-based approach, referred to as DPRLE in this proposal. This work is described briefly in Section 5.1 and appears in PLDI'09 [21].

2. A collaborative project, Hampi, that uses existing decision procedures to solve string constraints. This work appears in ISSTA'09 [24]. Our analysis of Hampi's scaling behavior inspired the decision to use a lazy search strategy. The Hampi encoding algorithm is not claimed in the proposed dissertation; we list it here for completeness.

3. An approach titled *Solving String Constraints Lazily* which is currently under submission. The approach is outlined in Section 5.2. We believe this work, in combination with (1), is suitable for a journal submission by early 2011.

4. A decision procedure for a mixed theory of string constraints and index operations. This work is scheduled for completion by the end of 2010. The approach is outlined in Section 5.3.

Time permitting, we may attempt one additional research thrust:

5. An investigation of proof engineering strategies for a certified string decision procedure. This project is described in Section 5.4, and tentatively scheduled for early 2011.

In the past, we have targeted venues like *Automated Software Engineering (ASE)*, *Computer Aided Verification (CAV)*, *International Symposium on Software Testing and Analysis (ISSTA)*, *Programming Language Design and Implementation (PLDI)*, and *Principles of Programming Languages (POPL)*. We propose targeting similar venues for the remaining work, together with a journal like *Transactions on Programming Languages and Systems (TOPLAS)*. The schedule includes limited slack for further collaborative projects not included in this proposal, including an industry internship scheduled for the summer of 2010. Finally, we will pro-actively seek out further collaborative efforts in target application areas, such as security and automated testing.

8 Conclusion

In this document, we made the case that program analyses could benefit from efficient general-purpose decision procedures that model common string library operations. We showed that existing approaches fall short when it comes to scalability, expressive utility, and correctness. We proposed the following research to address these issues:

1. We proposed a certified automaton-based algorithm, concat_intersect, and associated tool (*Decision Procedure for Regular Language Equations (DPRLE)*) for solving string constraints. This exploratory project demonstrates that string decision procedures can be used to generate attack inputs based on the output of a static bug finder [21]. This project also demonstrates that it is feasible to build constructive proofs of correctness for this class of algorithms.

2. Next, we turned our attention to scalability. We observed that existing algorithms often perform more work than is necessary to find a satisfying assignment. Based on this insight, we proposed a lazy search-based algorithm for solving constraints. Our preliminary results show this approach to be several orders of magnitude faster than the fastest existing implementation.

3. We proposed extending the feature set of (2), guided by a preliminary study of string function usage in real-world code. Our study showed that existing approaches do not directly support string indexing operations like `strstr`; these are commonly used in the PHP dataset that we examined. We hypothesized that it is feasible to solve string constraints and integer constraints integrally, without, for example, alternating between searching for a full set of integer solutions and searching for a satisfying assignment over strings.

4. Finally, we proposed an optional project on proving the total correctness of a string decision procedure, or, alternatively, to make an existing implementation self-certifying.

We will consider this work successful if each of the listed projects meets or exceeds the stated expectations. In the long term, we aim to create and maintain at least one general-purpose implementation of the proposed research to encourage adoption (and extension) by other researchers.

References

[1] S. Adams, T. Ball, M. Das, S. Lerner, S. K. Rajamani, M. Seigle, and W. Weimer. Speeding up dataflow analysis using flow-insensitive pointer analysis. In *SAS '02: Proceedings of the 9th International Symposium on Static Analysis*, pages 230–246, 2002.

[2] R. Axelsson, K. Heljanko, and M. Lange. Analyzing context-free grammars using an incremental sat solver. In *ICALP '08: Proceedings of the 35th international colloquium on Automata, Languages and Programming, Part II*, pages 410–422, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *TACAS '09: Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 307–321, 2009.

[4] H. Burch and R. C. Seacord. Programming language format string vulnerabilities c and c++ aren't alone when it comes to security vulnerability. In *http://www.ddj.com/security/197002914*, February 2007.

[5] B.-Y. E. Chang, A. J. Chlipala, and G. C. Necula. A framework for certified program analysis and its applications to mobile-code safety. In *VMCAI '06: Proceedings of the 7th Internation Conference on Verification, Model Checking, and Abstract Interpretation*, pages 174–189, 2006.

[6] K. Chen and D. Wagner. Large-scale analysis of format string vulnerabilities in debian linux. In *PLAS '07: Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security*, pages 75–84, 2007.

[7] A. Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 54–65, New York, NY, USA, 2007. ACM.

[8] A. Chlipala. A verified compiler for an impure functional language. In *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 93–106, New York, NY, USA, 2010. ACM.

[9] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *SAS '03: Proceedings of the 10th International Symposium on Static Analysis*, pages 1–18, 2003.

[10] T. Coquand and G. P. Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[11] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS '08: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[12] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.

[13] N. Eén and N. Sörensson. An extensible sat-solver. In *SAT '03: Proceedings of the 6th International Conference on the Theory and Applications of Satisfiability Testing*, pages 502–518, 2003.

[14] V. Ganesh, S. Berezin, and D. L. Dill. Deciding presburger arithmetic by model checking and comparisons with other methods. In *FMCAD '02: Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design*, pages 171–186, 2002.

[15] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *CAV '07: Proceedings of the 18th International Conference on Computer Aided Verification*, pages 519–531, 2007.

[16] P. Godefroid, A. Kieżun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, Tucson, AZ, USA, June 9–11, 2008.

[17] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[18] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS '08: Proceedings of the 15th Annual Symposium on Network Distributed Security Security*, 2008.

[19] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 526–538, 2002.

[20] K. J. Higgins. Cross-site scripting: attackers' new favorite flaw. Technical report, `http://www.darkreading.com/document.asp?doc_id=103774&WT.svl=news1_1`, Sept. 2006.

[21] P. Hooimeijer and W. Weimer. A decision procedure for subset constraints over regular languages. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 188–198, 2009.

[22] L. Ilie and S. Yu. Follow automata. *Information and Computation*, 186(1):140–162, 2003.

[23] S. Jha, S. A. Seshia, and R. Limaye. On the computational complexity of satisfiability solving for string theories. *CoRR*, abs/0903.2825, 2009.

[24] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: a solver for string constraints. In *ISSTA '09: Proceedings of the eighteenth international symposium on software testing and analysis*, pages 105–116, New York, NY, USA, 2009. ACM.

[25] K. Lakhotia, P. McMinn, and M. Harman. Handling dynamic data structures in search based testing. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1759–1766, July 2008.

[26] K. Lakhotia, P. McMinn, and M. Harman. Automated test data generation for coverage: Haven't we solved this problem yet? In *4th Testing Academia and Industry Conference - Practice and Research Techniques*, pages 95–104, Sept. 2009.

[27] R. Majumdar and R.-G. Xu. Directed test generation using symbolic grammars. In *ASE '07: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 134–143, 2007.

[28] P. Manual. Pcre; posix regex; strings. In *http://php.net/manual/en/book.strings.php*, December 2009.

[29] Y. Minamide. Static approximation of dynamically generated web pages. In *WWW '05: Proceedings of the 14th International Conference on the World Wide Web*, pages 432–441, 2005.

[30] Y. Minamide. Verified decision procedures on context-free grammars. In *TPHOLs '07: Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics*, pages 173–188, 2007.

[31] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *DAC '01: Proceedings of the 38th Conference on Design Automation*, pages 530–535, 2001.

[32] G. C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM.

[33] G. C. Necula. Translation validation for an optimizing compiler. In *PLDI '00: Proceedings of the 2000 ACM SIGPLAN conference on Programming language design and implementation*, pages 83–94, 2000.

[34] R. Richardson. FBI/CSI computer crime and security survey. Technical report, `http://www.gocsi.com/forms/csi_survey.jhtml`, 2008.

[35] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. Technical Report UCB/EECS-2010-26, EECS Department, University of California, Berkeley, Mar 2010.

[36] W. Security. Whitehat website security statistic report, 8th edition. `http://www.whitehatsec.com`, November 2009.

[37] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *POPL '06: Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–382, 2006.

[38] P. Thiemann. Grammar-based analysis of string expressions. In *Workshop on Types in Languages Design and Implementation*, pages 59–70, New York, NY, USA, 2005. ACM.

[39] M. Veanes, N. Bjørner, and L. de Moura. Solving extended regular constraints symbolically. Technical report, Microsoft Research, Dec. 2009.

[40] M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. Technical report, Microsoft Research, Oct. 2009.

[41] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 32–41, 2007.

[42] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering*, pages 364–374, 2009.

[43] Wu-ftpd. Bug: `http://www.cert.org/advisories/CA-2000-13.html`. Exploit: `http://www.securityfocus.com/bid/1387/exploit`, 2007.

[44] Y. Xie and A. Aiken. Saturn: A sat-based tool for bug detection. In K. Etessami and S. K. Rajamani, editors, *CAV '05: Proceedings of the 17th International Conference on Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 139–143. Springer, 2005.

[45] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, pages 179–192, July 2006.

[46] F. Yu, T. Bultan, and O. H. Ibarra. Symbolic string verification: Combining string analysis and size analysis. In *TACAS '09: Proceedings of the 15th Inernational Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2009.

Addendum

Ph.D. Dissertation Proposal
Pieter Hooimeijer
pieter@cs.virginia.edu

April 22, 2010

9 Related Work

We now briefly discuss work that is closely related to the proposed dissertation. Section 9.1 discusses a subset of the original work on string analyses (i.e., analyses that model string values) that include built-in models. Section 9.2 provides salient examples of the use of explicitly-external decision procedures by program analysis work. Finally, Section 9.3 gives an overview of decision procedures for strings, including more theoretically-oriented work on word equations.

9.1 Client Analyses

Christensen *et al.* first proposed a string analysis that soundly overapproximates string variables using regular languages [9]. This analysis uses a built-in model of strings using finite automata; a subset of that library is available separately from the analysis tool. Wassermann and Su [34] perform a similar static analysis to detect SQL injections. Their implementation extends Minamide's grammar-based analysis [24]. It statically models string values using context-free grammars, and detects potential database queries for which user input may change the intended syntactic structure of the query. In its original form, neither Wassermann and Su nor Minamide's analysis can generate example inputs.

In more recent work, Wassermann *et al.* show that many common string operations can be reversed using finite state transducers (FSTs) [35]. They use this method to generate inputs for SQL injection vulnerabilities in a concolic testing setup. Their algorithm is incomplete, however, and cannot be used to soundly rule out infeasible program paths. Yu *et al.* solve string constraints [37] for forward symbolic execution, using approximations ("widening automata") for non-monotonic operations, such as string replacement, to guarantee termination. Their approach has recently been extended to handle symbolic length constraints through the construction of length automata [38]. We note that these tools are for forward abstract interpretation; this task poses different challenges relative to constraint solving.

Godefroid *et al.* [13] use the SAGE architecture to perform guided random input generation (similar to previous work on random testcase generation by the same authors [14, 15]). It uses a grammar specification for valid program inputs rather than generating arbitrary input strings. This allows the analysis to reach beyond the program's input validation stages. Independent work by Majumdar and Xu [23] is similar to that of Godefroid *et al.*; CESE also uses symbolic execution to find inputs that are in the language of a grammar specification. All of these projects could

benefit from decision procedures for strings and regular expressions when performing symbolic execution, which requires decision procedures for strongest-postcondition calculations as well as ruling out infeasible paths.

9.2 The Use of Decision Procedures

Decision procedures have long been a fixture of program analyses. Typically a decision procedure handles queries over a certain *theory*, such as linear arithmetic, uninterpreted functions, boolean satisfiability [25, 36], pointer equality [26, 30], or bitwise operations and vectors [7, 12]. Nelson and Oppen presented a framework for allowing decision procedures to cooperate, forming an *automated theorem prover* to handle queries that span multiple theories and include first-order logic connectives [28]. In general, the *satisfiability modulo theories* problem is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic with equality. A number of SMT solvers, such as CVC [31] and Z3 [10], are available.

SLAM [4] and BLAST [17] are well-known examples of program analyses that make heavy use of external decision procedures: both are software model checkers that were originally written to call upon the Simplify theorem prover [11] to compute the effects of a concrete statement on an abstract model. This process, called predicate abstraction, is typically performed using decision procedures [22] and has led to new work in automated theorem proving [5]. SLAM has also made use of an explicit alias analysis decision procedure to improve performance [1]. BLAST uses proof-generating decision procedures to certify the results of model checking [16], just as they are used by proof-carrying code to certify code safety [27].

Another recent example is the EXE project [8], which combines symbolic execution and constraint solving [20] to generate user inputs that lead to defects. EXE has special handling for bit arrays and scalar values, and our work addresses an orthogonal problem. While a decision procedure for vectors might be used to model strings, merely reasoning about indexed accesses to strings of characters would not allow a program analysis to handle the high-level regular-expression checks present in many string-using programs. Note that the Hampi project [19] (a string constraint solver, listed below) and EXE share the same underlying decision procedure (STP [12]).

9.3 Theory and String Decision Procedures

There has been extensive theoretical work on language equations; Kunc provides an overview [21]. Work in this area has typically focused on complexity bounds and decidability results. Bala [3] defines the *Regular Language Matching (RLM)* problem, a generalization of the Regular Matching Assignments (RMA) problem [18] that allows both subset and superset constraints. Bala uses a construct called the *R-profile automaton* to show that solving RLM requires exponential space. Our proposed decision procedures supports a different set of operations (e.g., we do not allow Kleene $\star$ on variables).

Bjørner *et al.* present a decision procedure for several common string operations by reduction to existing SMT theories [6], fixing the string lengths in a separate step. They show that the addition of a replace function makes the theory undecidable. In concurrent work, we show that bounded context-free language constraints can be solved efficiently by direct conversion to SAT. The Hampi tool [19] is a solver for string constraints over fixed-size string variables. It supports

regular languages, fixed-size context-free languages, and a inclusion constraints. Hampi has been extensively evaluated in static and dynamic analysis tools and for automatic test generation.

In recent work, Saxena *et al.* provide an extension to Hampi that supports multiple variables and length constraints [29]; this is a subset of the integer index operations defined in this proposal. The associated tool, Kaluza, builds on Hampi's underlying solver (STP [12]) to iteratively solve integer constraints (by calling STP directly) and string constraints (using Hampi). We hypothesize that a more specialized approach (as proposed for this dissertation) would yield better scalabililty.

The CFG Analyzer tool [2] is a solver for bounded versions of otherwise-undecidable context-free language problems. Problems such as inclusion, intersection, universality, equivalence and ambiguity are handled via a reduction to satisfiability for propositional logic in the bounded case. The Rex tool [32, 33] solves string constraints through a symbolic encoding of finite state automata into Z3 SMT solver [10]. An important benefit of this strategy is that string constraints can be readily integrated with other theories (e.g., linear arithmetic) handled by Z3. A disadvantage is that the encoding is relatively inefficient; in our preliminary results we showed that Hampi and our prototype consistently out-performed Rex by up to four orders of magnitude.

The DPRLE tool [18] is a decision procedure for regular language constraints, such as those involving concatenation and intersection operators between multiple variables. The tool focuses on generating entire sets of satisfying assignments rather than single strings: often constraints over multiple variables can yield multiple disjoint solution sets. The core algorithm of DPRLE has been formally proved correct in a constructive logic framework. More recent approaches have focused on producing string assignments (treating the variables as strings rather than sets), since the ultimate goal of many client analyses is to produce single string witnesses.

References

[1] S. Adams, T. Ball, M. Das, S. Lerner, S. K. Rajamani, M. Seigle, and W. Weimer. Speeding up dataflow analysis using flow-insensitive pointer analysis. In *SAS '02: Proceedings of the 9th International Symposium on Static Analysis*, pages 230–246, 2002.

[2] R. Axelsson, K. Heljanko, and M. Lange. Analyzing context-free grammars using an incremental sat solver. In *ICALP '08: Proceedings of the 35th international colloquium on Automata, Languages and Programming, Part II*, pages 410–422, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] S. Bala. Regular language matching and other decidable cases of the satisfiability problem for constraints between regular open terms. In *STACS*, pages 596–607, 2004.

[4] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *European Systems Conference*, pages 103–122, Apr. 2006.

[5] T. Ball, B. Cook, S. K. Lahiri, and L. Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. In *Computer Aided Verification*, pages 457–461, 2004.

[6] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *TACAS '09: Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 307–321, 2009.

[7] R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. Brady. Deciding bit-vector arithmetic with abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 358–372, 2007.

[8] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *Computer and Communications Security*, pages 322–335, 2006.

[9] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *SAS '03: Proceedings of the 10th International Symposium on Static Analysis*, pages 1–18, 2003.

[10] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS '08: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[11] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.

[12] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *CAV '07: Proceedings of the 18th International Conference on Computer Aided Verification*, pages 519–531, 2007.

[13] P. Godefroid, A. Kieżun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, Tucson, AZ, USA, June 9–11, 2008.

[14] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[15] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS '08: Proceedings of the 15th Annual Symposium on Network Distributed Security Security*, 2008.

[16] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 526–538, 2002.

[17] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Principles of Programming Languages*, pages 58–70, 2002.

[18] P. Hooimeijer and W. Weimer. A decision procedure for subset constraints over regular languages. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 188–198, 2009.

[19] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: a solver for string constraints. In *ISSTA '09: Proceedings of the eighteenth international symposium on software testing and analysis*, pages 105–116, New York, NY, USA, 2009. ACM.

[20] J. Kodumal and A. Aiken. Banshee: A scalable constraint-based analysis toolkit. In *Static Analysis Symposium*, pages 218–234, 2005.

[21] M. Kunc. What do we know about language equations? In *Developments in Language Theory*, pages 23–27, 2007.

[22] S. K. Lahiri, T. Ball, and B. Cook. Predicate abstraction via symbolic decision procedures. *Logical Methods in Computer Science*, 3(2), 2007.

[23] R. Majumdar and R.-G. Xu. Directed test generation using symbolic grammars. In *ASE '07: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 134–143, 2007.

[24] Y. Minamide. Static approximation of dynamically generated web pages. In *WWW '05: Proceedings of the 14th International Conference on the World Wide Web*, pages 432–441, 2005.

[25] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *DAC '01: Proceedings of the 38th Conference on Design Automation*, pages 530–535, 2001.

[26] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *Principles of Programming Languages*, pages 327–338, 2007.

[27] G. C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM.

[28] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[29] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. Technical Report UCB/EECS-2010-26, EECS Department, University of California, Berkeley, Mar 2010.

[30] B. Steensgaard. Points-to analysis in almost linear time. In *Principles of Programming Languages*, pages 32–41, 1996.

[31] A. Stump, C. W. Barrett, and D. L. Dill. Cvc: A cooperating validity checker. In *Computer Aided Verification*, pages 500–504, 2002.

[32] M. Veanes, N. Bjørner, and L. de Moura. Solving extended regular constraints symbolically. Technical report, Microsoft Research, Dec. 2009.

[33] M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. Technical report, Microsoft Research, Oct. 2009.

[34] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 32–41, 2007.

[35] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *International Symposium on Software testing and analysis*, pages 249–260, 2008.

[36] Y. Xie and A. Aiken. Saturn: A sat-based tool for bug detection. In K. Etessami and S. K. Rajamani, editors, *CAV '05: Proceedings of the 17th International Conference on Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 139–143. Springer, 2005.

[37] F. Yu, T. Bultan, M. Cova, and O. H. Ibarra. Symbolic string verification: An automata-based approach. In *SPIN '08: Proceedings of the 15th international workshop on Model Checking Software*, pages 306–324, Berlin, Heidelberg, 2008. Springer-Verlag.

[38] F. Yu, T. Bultan, and O. H. Ibarra. Symbolic string verification: Combining string analysis and size analysis. In *TACAS '09: Proceedings of the 15th Inernational Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2009.