Statically Typed String Sanitation Inside a Python

Nathan Fulton Cyrus Omar Jonathan Aldrich
Carnegie Mellon University
Pittsburgh, PA
{nathanfu, comar, aldrich}@cs.cmu.edu

ABSTRACT

Web applications must ultimately command systems like web browsers and database engines using strings. Strings constructed using user input that has not been properly sanitized can thus cause command injection vulnerabilities. In this paper, we introduce *regular string types*, which classify strings known statically to be in a specified regular language. These types come equipped with common operations like concatenation, substitution and coercion, so they can be used to implement, in essentially a conventional manner, the parts of a web application or application framework that construct command strings. Simple type annotations at key interfaces can be used to statically verify that sanitization has been performed correctly without introducing redundant run-time checks. We specify this type system as a minimal typed lambda calculus, λ_{RS}.

To be practical, adopting a specialized type system like this should not require the adoption of a new programming language. Instead, we favor extensible type systems: new type system fragments like this should be distributed as libraries atop a mechanism that guarantees that they can be safely composed. We support this by 1) specifying a translation from λ_{RS} to a language containing only strings and regular expressions, then, taking Python as such a language, 2) implement the type system together with the translation as a library using `atlang`, an extensible static type system for Python (being developed by the authors).

1. INTRODUCTION

Command injection vulnerabilities are among the most common and severe security vulnerabilities in modern web applications [10]. They arise because web applications, at their boundaries, control external systems by issuing commands represented using strings. For example, web browsers are controlled using HTML and Javascript sent from a server as a string, and database engines execute SQL queries also sent as strings. When these commands must include data derived from user input, care must be taken to ensure that the user cannot construct input that subverts the intended command. For example, a SQL query constructed using string concatenation exposes a SQL injection vulnerability:

```
'SELECT * FROM users WHERE name="' + name + '"'
```

If a malicious user enters the name `'"; DROP TABLE users --'`, the entire database could be erased.

To avoid this problem, the program must *sanitize* user input. For example, in this case, the developer (or, more often, a framework) might define a function `sanitize` that prepends double quotes (and existing backslashes) with a backslash, which SQL treats safely. Guaranteeing that user input has already been sanitized before it is used to construct a command is challenging. Note that this function is not idempotent, so it should only be called once.

We observe that most such sanitization techniques can be understood in terms of *regular languages* [5]. For example, `name` should be a string in the language described by the regular expression `([^"\]|(\")|(\\))*` – a sequence of characters other than quotation marks and backslashes; these can only appear escaped. This concrete pattern syntax can be understood to desugar, in a standard way, to the syntax for regular expressions shown in Figure 1, where $r \cdot r$ is sequencing and $r + r$ is disjunction. We will work with this "core" for simplicity.

In this paper, we present a static type system that tracks the regular language a string belongs to. For example, the output of `sanitize` will be in the regular language described by the regular expression above. We write this as $\mathcal{L}\{r\}$. We take advantage of closure and decidability properties of regular languages to support a number of useful operations on values of such *regular string types*, including *replacement* of substrings matching a given regular expression. These make it possible to implement sanitation protocols like the one just described in an essentially conventional manner. The result is a system where the fact that a string has been correctly sanitized is manifest in its type. Missing calls to sanitization functions are detected statically, and, importantly, so are *incorrectly implemented sanitization functions* (i.e. these functions need not be trusted). These guarantees require run-time checks only when going from less precise to more precise types (e.g. at the edges of the system, where user input has not yet been validated).

We will begin in Sec. 2 by specifying this type system minimally, as a conservative extension of the simply typed lambda calculus called λ_{RC}. This allows us to specify the guarantees that the type system provides precisely. We also formally specify a translation from this calculus to a typed calculus with only standard strings and regular expressions, intending it as a guide to language implementors interested

$$r \;::=\; \epsilon \mid . \mid a \mid r \cdot r \mid r + r \mid r* \qquad a \in \Sigma$$

Figure 1: Regular expressions over the alphabet Σ.

$$\begin{array}{lll} \sigma & ::= & \sigma \to \sigma & \text{source types} \\ & \mid & \mathsf{stringin}[r] & \\ & & & \\ e & ::= & x & \text{source terms} \\ & \mid & \lambda x{:}\sigma.e & \\ & \mid & e(e) & \\ & \mid & \mathsf{rstr}[s] & s \in \Sigma^* \\ & \mid & \mathsf{rconcat}(e,e) & \\ & \mid & \mathsf{rreplace}[r](e,e) & \\ & \mid & \mathsf{rcoerce}[r](e) & \end{array}$$

Figure 2: Syntax of λ_{RS}.

in building this feature into their own languages. This also demonstrates that no additional space overhead is required.

Waiting for a language designer to build this feature in is unsatisfying. Moreover, we would also face a "chicken-and-egg problem": justifying its inclusion into a commonly used language requires empirically demonstrating that it is useful, but this is difficult to do if developers have no way to use it in practice. As such, we take the position that a better path forward for the community is to work within a programming language where such type system fragments can be introduced modularly and orthogonally, as libraries.

In Sec. 3, we show how to implement the type system fragment we have specified using `atlang`, an extensible static type system implemented as a library inside Python. `atlang` leverages local type inference to control the semantics of literal forms, so reguar string types can be introduced using string literals without any run-time overhead. Coercions that are known to be safe due to a sublanguage relationship are performed implicitly, also without run-time overhead. This results in a *usably secure* system: working with regular strings differs little from working with standard strings.

We conclude after discussing related work in Sec. **??**.

2. REGULAR STRING TYPES, MINIMALLY

In this section, we define a minimal typed lambda calculus with regular string types called λ_{RS}. This will serve as the source language for our translation to a calculus with only standard strings and regular expressions, defined in Sec. **??**.

The syntax of λ_{RS} is given in Figure 2, its static semantics are specified in Figure 3 and its evaluation semantics are specified in Figure 4.

The system has regular expression types $\mathsf{stringin}[r]$ whre r is a regular expression. Expressions of this type evaluate to string literals in the language described by r. Operations on expressions of type $\mathsf{stringin}[r]$ preserve this property.

The premier operation for manipulating strings in λ_S is string substitution, which is a familiar operation to any programmer who has used regular expressions. The replacement operation replaces all instances of a pattern in one string with another string; for instance, $\texttt{lsubst}(a|b, a, c) = c$. In order to compute the type resulting from substitution, we also need to compute the result of replacing one language with another inside a given language. Finally, just for convenience, we provide a coerce operation. The introduction of coercion requires handling of runtime errors.

$\boxed{\Psi \vdash S : \sigma}$ $\quad \Psi ::= \emptyset \mid \Psi, x : \sigma$

$$\text{S-T-StringIn-I}\ \frac{s \in \mathcal{L}\{r\}}{\Psi \vdash \mathsf{rstr}[s] : \mathsf{stringin}[r]}$$

$$\text{S-T-Concat}\ \frac{\Psi \vdash S_1 : \mathsf{stringin}[r_1] \qquad \Psi \vdash S_2 : \mathsf{stringin}[r_2]}{\Psi \vdash \mathsf{rconcat}(S_1, S_2) : \mathsf{stringin}[r_1 \cdot r_2]}$$

$$\text{S-T-Replace}\ \frac{\Psi \vdash S_1 : \mathsf{stringin}[r_1] \qquad \Psi \vdash S_2 : \mathsf{stringin}[r_2] \qquad \texttt{lreplace}(r, r_1, r_2) = r'}{\Psi \vdash \mathsf{rreplace}[r](S_1, S_2) : \mathsf{stringin}[r']}$$

$$\text{S-T-coerce}\ \frac{\Psi \vdash S : \mathsf{stringin}[r']}{\Psi \vdash \mathsf{rcoerce}[r](S) : \mathsf{stringin}[r]}$$

Figure 3: Typing rules for our fragment of λ_S. The typing context Ψ is standard.

$\boxed{S \Downarrow S}$ $\boxed{S\ \mathsf{err}}$

$$\text{S-E-RStr}\ \frac{}{\mathsf{rstr}[s] \Downarrow \mathsf{rstr}[s]} \qquad \text{S-E-Concat}\ \frac{S_1 \Downarrow \mathsf{rstr}[s_1] \qquad S_2 \Downarrow \mathsf{rstr}[s_2]}{\mathsf{rconcat}(S_1, S_2) \Downarrow \mathsf{rstr}[s_1 s_2]}$$

$$\text{S-E-Replace}\ \frac{S_1 \Downarrow \mathsf{rstr}[s_1] \qquad S_2 \Downarrow \mathsf{rstr}[s_2] \qquad \texttt{lsubst}(r, s_1, s_2) = s}{\mathsf{rreplace}[r](S_1, S_2) \Downarrow \mathsf{rstr}[s]}$$

$$\text{S-E-Coerce-OK}\ \frac{S \Downarrow \mathsf{rstr}[s] \qquad s \in \mathcal{L}\{r\}}{\mathsf{rcoerce}[r](S) \Downarrow \mathsf{rstr}[s]} \qquad \text{S-E-Coerce-Err}\ \frac{S \Downarrow \mathsf{rstr}[s] \qquad s \notin \mathcal{L}\{r\}}{\mathsf{rcoerce}[r](S)\ \mathsf{err}}$$

Figure 4: Big step semantics for our fragment of λ_S. Error propagation rules are omitted.

The underlying language λ_P has only one type for strings. We prove that whenever a term is translated from λ_S to λ_P, correctness is preserved. The only exception is in the case of unsafe casts in λ_S, which are unnecessary but are included to demonstrate that the regex library of λ_P may be used to insert dynamic checks whenever even when developers are not careful about using statically checked operations.

A brief outline of this section follows:

- Page 3 contains a definition of λ_S, λ_P and the translation from λ_S to λ_P. Grammar follows immediately at the top of page 4.
- In §2.1 we state some properties about regular expressions which are needed in our correctness proofs.
- In §2.2 we prove type safety for λ_P as well as both type safety and correctness for λ_S.
- In §2.3 we prove that translation preserves the correctness result about λ_S.

$\boxed{[\![S]\!] = P}$

$$\textsc{Tr-string}\ \frac{}{[\![\mathsf{rstr}[s]]\!] = \mathsf{str}[s]} \qquad \textsc{Tr-Concat}\ \frac{[\![S_1]\!] = P_1 \qquad [\![S_2]\!] = P_2}{[\![\mathsf{rconcat}(S_1, S_2)]\!] = \mathsf{concat}(P_1, P_2)} \qquad \textsc{Tr-Subst}\ \frac{[\![S_1]\!] = P_1 \qquad [\![S_2]\!] = P_2}{[\![\mathsf{rreplace}[r](S_1, S_2)]\!] = \mathsf{replace}(\mathsf{rx}[r], P_1, P_2)}$$

$$\textsc{Tr-Coerce-Ok}\ \frac{S : \mathsf{rstr}[r] \qquad \mathcal{L}\{r'\} \subseteq \mathcal{L}\{r\}}{[\![\mathsf{rcoerce}[r'](S)]\!] = \mathsf{str}[s]} \qquad \textsc{Tr-Coerce-NotOk}\ \frac{[\![S]\!] = P \qquad S : \mathsf{rstr}[r] \qquad \mathcal{L}\{r'\} \not\subseteq \mathcal{L}\{r\}}{[\![\mathsf{rcoerce}[r'](S)]\!] = \mathsf{check}(\mathsf{rx}[r'], P)}$$

Figure 9: Translation from source terms (S) to target terms (P). The translation is type-directed in the Tr-Coerce cases.

$\boxed{\Theta \vdash P : \theta}$ $\quad \Theta ::= \emptyset \mid \Theta, x : \theta$

$$\textsc{P-T-String}\ \frac{}{\Theta \vdash \mathsf{str}[s] : \mathsf{string}} \qquad \textsc{P-T-Regex}\ \frac{}{\Theta \vdash \mathsf{rx}[r] : \mathsf{regex}}$$

$$\textsc{P-T-Concat}\ \frac{\Theta \vdash P_1 : \mathsf{string} \qquad \Theta \vdash P_2 : \mathsf{string}}{\Theta \vdash \mathsf{concat}(P_1, P_2) : \mathsf{string}}$$

$$\textsc{P-T-Replace}\ \frac{\Theta \vdash P_1 : \mathsf{regex} \qquad \Theta \vdash P_2 : \mathsf{string} \qquad \Theta \vdash P_3 : \mathsf{string}}{\Theta \vdash \mathsf{preplace}(P_1, P_2, P_3) : \mathsf{string}}$$

$$\textsc{P-T-Check}\ \frac{\Theta \vdash P_1 : \mathsf{regex} \qquad \Theta \vdash P_2 : \mathsf{string}}{\Theta \vdash \mathsf{check}(P_1, P_2) : \mathsf{string}}$$

Figure 5: Typing rules for our fragment of λ_P. The typing context Θ is standard.

$\boxed{P \Downarrow P}$ $\boxed{P\ \mathsf{err}}$

$$\textsc{P-E-Str}\ \frac{}{\mathsf{str}[s] \Downarrow \mathsf{str}[s]} \qquad \textsc{P-E-Rx}\ \frac{}{\mathsf{rx}[r] \Downarrow \mathsf{rx}[r]} \qquad \textsc{P-E-Concat}\ \frac{P_1 \Downarrow \mathsf{str}[s_1] \qquad P_2 \Downarrow \mathsf{str}[s_2]}{\mathsf{concat}(P_1, P_2) \Downarrow \mathsf{str}[s_1 s_2]}$$

$$\textsc{P-E-Replace}\ \frac{P_1 \Downarrow \mathsf{rx}[r] \qquad P_2 \Downarrow \mathsf{str}[s_2] \qquad P_3 \Downarrow \mathsf{str}[s_3] \qquad \mathtt{lsubst}(r, s_2, s_3) = s}{\mathsf{preplace}(P_1, P_2, P_3) \Downarrow \mathsf{str}[s]}$$

$$\textsc{P-E-Check-OK}\ \frac{P_1 \Downarrow \mathsf{rx}[r] \qquad P_2 \Downarrow \mathsf{rstr}[s] \qquad s \in \mathcal{L}\{r\}}{\mathsf{check}(P_1, P_2) \Downarrow \mathsf{str}[s]}$$

$$\textsc{P-E-Check-Err}\ \frac{P_1 \Downarrow \mathsf{rx}[r] \qquad P_2 \Downarrow \mathsf{str}[s] \qquad s \notin \mathcal{L}\{r\}}{\mathsf{check}(P_1, P_2)\ \mathsf{err}}$$

Figure 6: Big step semantics for our fragment of λ_P. Error propagation rules are omitted.

θ ::= $\theta \to \theta$ target types
| string
| regex

P ::= $\lambda x.e$ target terms
| ee
| str$[s]$
| rx$[r]$
| concat(P, P)
| preplace(P, P, P)
| check(P, P)

Figure 7: Syntax for the fragment of our target language, λ_P, containing strings and statically constructed regular expressions.

2.1 Definition of λ_S

The λ_S system extends the simply-typed lambda calculus with regular expression types; an explanation of significant typing rule follows:

- Rule S-T-Concat states that the type of concatenated strings is obtained by concatenating the regular expressions for each string.
- Rule S-T-Replace defines the type of rreplace. The expression rreplace evaluates to the result of substituting every string matching r in s_1 with s_2. This operation corresponds to the str_replace function of PHP. The type of these expressions is defined in terms of an extra-linguistic lreplace function, which is defined later in this section. The expression $\texttt{lreplace}(r, r_1, r_2)$ is obtained by starting with r_1, and replacing any subexpression matching r with r_2.
- Rule S-T-Coerce allows coercion; however, note that this coercion cannot invalidate our safety property because we ensure appropraite checks are always inserted wherever coercions are used.

2.2 Definition of λ_P

The system λ_P is a straight-forward extension of a simply typed lambda calculus with a string type and a regular expression tpye. We include two operations which are available in the regular expression library of any modern programming language. The check operation ensures that an expression recognizes a string, and the replace operation is string replacement.

2.3 Definition of Translation

The translation from λ_S to λ_P is defined in figure 5. The coercion cases are most interesting. If the safety of coercion in manifest in the types of the expressions, then no runtime check is inserted. If the safety of coercion is not manifest in the types, then a check is inserted.

In practice, the type of a replacement rarely matches a specification. Therefore, it is convienant in an implementation to always insert the appropriate coercion, and then only raise type errors when an automatically inserted coercion actually requires the insertion of a runtime check. Alternatively, this policy may be codified in the type system itself using subtyping [2].

2.4 Properties of Regular Languages

Our type safety proof for language S replies on a relationship between string substitution and language substitution given in lemma 5. We also rely upon several other properties of regular languages. Throughout this section, we fix an alphabet Σ over which strings s and regular expressions r are defined. throughout the paper, $\mathcal{L}\{r\}$ refers to the language recognized by the regular expression r. This distinction between the regular expression and its language – typically elided in the literature – makes our definition and proofs about systems S and P more readable.

Lemma 1. *Properties of Regular Languages and Expressions. The following are properties of regular expressions which are necessary for our proofs: If $s_1 \in \mathcal{L}\{r_1\}$ and $s_2 \in \mathcal{L}\{r_2\}$ then $s_1 s_2 \in \mathcal{L}\{r_1 r_2\}$. For all strings s and regular expressions r, either $s \in \mathcal{L}\{r\}$ or $s \notin \mathcal{L}\{r\}$. Regular languages are closed under difference, right quotient, reversal, and string homomorphism.*

If any of these properties are unfamiliar, the reader may refer to a standard text on the subject [5].

Definition 2 (lsubst)**.** The relation $\texttt{lsubst}(r, s_1, s_2) = s$ produces a string s in which all substrings of s_1 matching r are replaced with s_2.

Definition 3 (lreplace)**.** The relation $\texttt{lreplace}(r, r_1, r_2) = r'$ relates r, r_1, and r_2 to a language r' containing all strings of r_1 except that any substring $s_{pre} s s_{post} \in \mathcal{L}\{r_1\}$ where $s \in \mathcal{L}\{r\}$ is replaced by the set of strings $s_{pre} s_2 s_{post}$ for all $s_2 \in \mathcal{L}\{r_2\}$ (the prefix and postfix positions may be empty).

Lemma 4. *Closure. If $\mathcal{L}\{r\}, \mathcal{L}\{r_1\}$ and $\mathcal{L}\{r_2\}$ are regular expressions, then $\mathcal{L}\{\texttt{lreplace}(r, r_1, r_2)\}$ is also a regular language.*

Proof. The theorem follows from closure under difference, right quotient, reversal and string homomorphism. □

Lemma 5. *Substitution Correspondence. If $s_1 \in \mathcal{L}\{r_1\}$ and $s_2 \in \mathcal{L}\{r_2\}$ then $\texttt{lsubst}(r, s_1, s_2) \in \mathcal{L}\{\texttt{lreplace}(r, s_1, s_2)\}$.*

Proof. The theorem follows from the definitions of lsubst and lreplace; note that language substitutions over-approximate string substitutions. □

2.5 Safety of the Source and Target Languages

In this section, we establish type safety for the source (λ_S) and target (λ_P) languages. In addition to type safety, we also prove a stronger correctness property for λ_S.

Our first two theorems establish that our rules preserve the well-formedness of regular expressions. The standard lemmas required for safety of the simply typed lambda clauculus are also required; proofs for these do not differ substntially from [4].

Lemma 6. *If $\Psi \vdash S$: stringin$[r]$ then r is a well-formed regular expression.*

Proof. The only non-trivial case is S-T-Replace, which follows from lemma 4. □

Lemma 7. *If $\Theta \vdash P$: regex then $P \Downarrow$ rx$[r]$ such that r is a well-formed regular expression.*

We now prove safety for the string fragment of the source and target languages. The easiest property is type safety of λ_P, which follows almost directly from type safety for the simply typed lambda calculus.

Theorem 8. *Let P be a term in the target language. If $\Theta \vdash P : \theta$ then $P \Downarrow P'$ and $\Theta \vdash P' : \theta$, or else P err.*

Safety for the string fragment is more involved, because it involves validating that the type system's definition is justified by our theorems about regular languages. Note that type safety does *not* guarantee that strings of a type are in the correct language. We isolate this property so that we may reason about its preservation under translation to λ_P, which does not have regular expression types.

Theorem 9. *Type Safety for the String Fragment of λ_S. Let S be a term in the source language. If $\Psi \vdash S : \mathsf{stringin}[r]$ then $S \Downarrow \mathsf{rstr}[s]$ and $\Psi \vdash \mathsf{rstr}[s] : \mathsf{stringin}[r]$; or else S err.*

Proof. By induction on the typing relation, where (a) case holds by lemma 1 in the S-T-Concat case and lemma 5 in the S-T-Replace case.. The (b) cases hold by unstated, but standard, error propagation rules. □

In addition to safety, λ_S requires a correctness result ensuring that well-typed terms of regular string type are in the language associated with their type.

Theorem 10. *Correctness of Input Sanitation for λ_S. If $\Psi \vdash S : \mathsf{stringin}[r]$ and $S \Downarrow \mathsf{rstr}[s]$ then $s \in \mathcal{L}\{r\}$.*

Proof. Follows directly from type safety, canonical forms for λ_S. □

2.6 Translation Correctness

The main theorem of this paper is Theorem 12, which establishes that Theorem 10 is preserved under translation into the target language λ_P.

Establishing this result requires an additional theorem establishing a relationship between canonical forms for the string fragments of λ_S and λ_P.

Theorem 11. *Translation Correctness. If $\Psi \vdash S : \mathsf{stringin}[r]$ then there exists a P such that $[\![S]\!] = P$ and either: (a) $P \Downarrow \mathsf{str}[s]$ and $S \Downarrow \mathsf{rstr}[s]$, or (b) P err and S err.*

Proof. The proof proceeds by induction on the typing relation for S and an appropriate choice of P; in each case, the choice is obvious. The subcases (a) proceed by inversion and appeals to our type safety theorems as well as the induction hypothesis. The subcases (b) proceed by the standard error propagation rules omitted for space. Throughout the proof, properties from the closure lemma for regular languages are necessary. □

Finally, our main theorem establishes that input sanitation correctness of λ_S is preserved under the translation into λ_P.

Theorem 12. *Correctness of Input Sanitation for Translated Terms. If $[\![S]\!] = P$ and $\Psi \vdash S : \mathsf{stringin}[r]$ then either P err or $P \Downarrow \mathsf{str}[s]$ for $s \in \mathcal{L}\{r\}$.*

Proof. By theorem 11, $P \Downarrow \mathsf{str}[s]$ implies that $S \Downarrow \mathsf{rstr}[s]$. By theorem 10, the above property together with the assumption that S is well-typed implies that $s \in \mathcal{L}\{r\}$. □

```
@fn
def sanitize(s : string_in[r'.*']):
  return (s.replace(r'"', '&quot;')
           .replace(r'<', '&lt;')
           .replace(r'>', '&gt;'))

@fn
def results_query(s : string_in[r'[^"]*']):
  return 'SELECT * FROM users WHERE name="' + s + '"'

@fn
def results_div(s : string_in[r'[^<>]*']):
  return '<div>Results for ' + s + '</div>'

def main(db):
  input = sanitize(user_input())
  results = db.execute(results_query(input))
  return results_div(input) + format(results)
```

Figure 8: Regular string types in atlang, a library that enables static type checking for Python.

3. IMPLEMENTATION IN ATLANG

3.1 An Example

Figure 3 demosntrates the use of string types in atlang. The sanitize function takes an arbitrary string and returns a string without double quotes or left and right brackets. In this example, we use HTML escape sequences.

The main function receives user input and passes this input to a sanitize function, which replaces all double quotes and brackets with HTML escape sequences.

The result of applying sanitize to input is appended to two functions which construct a safe query and safe output. The arguments to the result and output construction functions constitute *specifications*. In the case of results_query, this specification ensures that user input is always interpreted as a string literal by the SQL server. In the case of results_div, this specification ensures that user input does not contain any HTML tags, which is a conservative but effective policy for preventing XSS attacks.

Note that input does not actually meet these specifications without additional machinery. The type of input is quite large and does not actually equal the specified domains of the query or output construction methods. This mismatch is common – in fact, nearly universal. Therefore, our implementation includes a simple subtyping relation between regular expression types. This subtyping relation is justified theoretically by the fact that language inclusion is decidable; see [2] for a formal definition of the subtyping relation. Additionally, our extension remains composable because subtyping is defined on a type-by-type basis; see [3] for a discussion of subtyping in Atlang (referred to there as Ace).

3.2 Implementation of the Regular Expression Type

We implemented a variation on the type system presented in this paper. The only significant difference is that we only support replacements where s_2 is the empty string. Therefore, our implementation respects the system presented in section 2 only modulo the definition of lreplace.

Atlang translates programs using type definitions, which may extend both the static and dynamic semantics of the language. New types are defined as Python classes; figure 3 contains the source code of our implementation.

The string_in type has an indexing regular expression `idx`.

```
class string_in(atlang.Type):
  def __init__(self, rx):
    rx = rx_normalize(rx)
    atlang.Type.__init__(idx=rx)

  def ana_Str(self, ctx, node):
    if not in_lang(node.s, self.idx):
      raise atlang.TypeError("...", node)

  def trans_Str(self, ctx, node):
    return astx.copy(node)

  def syn_BinOp_Add(self, ctx, node):
    left_t = ctx.syn(node.left)
    right_t = ctx.syn(node.right)
    if isinstance(left_t, string_in):
      left_rx = left_t.idx
      if isinstance(right_t, string_in):
        right_rx = right_t.idx
        return string_in[lconcat(left_rx, right_rx)]
    raise atlang.TypeError("...", node)

  def trans_BinOp_Add(self, ctx, node):
    return astx.copy(node)

  def syn_Method_replace(self, ctx, node):
    [rx, exp] = node.args
    if not isinstance(rx, ast.Str):
      raise atlang.TypeError("...", node)
    rx = rx.s
    exp_t = ctx.syn(exp)
    if not isinstance(exp_t, string_in):
      raise atlang.TypeError("...", node)
    exp_rx = exp_t.idx
    return string_in[lreplace(self.idx, rx, exp_rx)]

  def trans_Method_replace(self, ctx, node):
    return astx.quote(
      """__import__(re); re.sub(%0, %1, %2)""",
      astx.Str(s=node.args[0]),
      astx.copy(node.func.value),
      astx.copy(node.args[1]))

  def syn_Method_check(self, ctx, node):
    [rx] = node.args
    if not isinstance(rx, ast.Str):
      raise atlang.TypeError("...", node)
    return string_in[rx.s]

  def trans_Method_check(self, ctx, node):
    return astx.quote(
      """__import__(string_in_helper);
      string_in_helper.coerce(%0, %1)""",
      astx.Str(s=other_t.idx),
      astx.copy(node))

  def check_Coerce(self, ctx, node, other_t):
    # coercions can only be defined between
    # types with the same type constructor,
    if rx_sublang(other_t.idx, self.idx):
      return other_t
    else: raise atlang.TypeError("...", node)
```

Figure 9: Implementation of the `string_in` type constructor in atlang.

```
output of successful compilation
```

Figure 10: Output of successful compilation.

```
output of failed compilation
```

Figure 11: Output of failed compilation.

Our translation is defined by the trans_ methods while the syn_ methods define our type checker. Atlang defers type checking and translation to these methods whenever an expression of type string_in is encountered.

3.3 Related Work and Alternative Approaches

The input sanitation problem is well-understood. There exist a large number of techniques and technologies, proposed by both practitioners and researchers, for preventing injection-style attacks. In this section, we explain how our approach to the input sanitation problem differs from each of these approaches. More important than these differences, however, is our more general assertion that language extensibility is a promising approach toward consideration of security goals in programming language design.

Unlike *frameworks and libraries* provided by languages such as Haskell and Ruby, our type system provides a *static* guarantee that input is always properly sanitized before use. Doing so requires reasoning about the operations on regular languages corresponding to standard operations on strings; we are unaware of any production system which contains this form of reasoning. Therefore, even where frameworks and libraries provide a viable interface or wrapper around input sanitation, our approach is complementary because it ensures the correctness of the framework or library itself. Furthermore, our approach is more general than database abstraction layers because our mechanism is applicable to all forms of command injection (e.g. shell injection or remote file inclusion).

A number of research languages provide static guarantees that a program is free of input sanitation vulnerabilities [1]. Unlike this work, our solution to the input sanitation problem has a very low barrier to adoption; for instance, our implementation conservatively extends Python – a popular language among web developers. We also believe our general approach is better-positioned for security, where continuously evolving threats might require frequent addition of new analyses; in these cases, the composability and generality of our approach is a substantial advantage.

The Wyvern programming language provides a general framework for composing language extensions [9][8]. Our work identifies one particular extension, and is therefore complementary to Wyvern and related work on extensible programming languages. We are also unaware of any extensible programming languages which emphasize applications to security concerns.

Incorporating regular expressions into the type system is not novel. The XDuce system [7, 6] checks XML documents against schema using regular expressions. Similarly, XHaskell [11] focuses on XML documents. We differ from this and related work in at least three ways:

- Our system is defined within an extensible type system.
- We demonstrate that regular expression types are applicable to the web security domain, whereas previous work on regular expression types focused on XML schema.
- Although our static replacement operation is definable in some languages with regular expression types, we are the first to expose this operation and connect the semantics of regular language replacement with the se-

mantics of string substitution via a type safety and compilation correctness argument.

In conclusion, our contribution is a type system, implemented within an extensible type system, for checking the correctness of input sanitation algorithms.

4. CONCLUSION

Composable analyses which complement existing approaches constitute a promising approach toward the integration of security concerns into programming languages. In this paper, we presented a system with both of these properties and defined a security-preserving transformation. Unlike other approaches, our solution complements existing, familiar solutions while providing a strong guarantee that traditional library and framework-based approaches are implemented and utilized correctly.

5. REFERENCES

[1] A. Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *OSDI'10: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2010.

[2] N. Fulton. Security through extensible type systems. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 107–108, New York, NY, USA, 2012. ACM.

[3] N. Fulton. A typed lambda calculus for input sanitation. Undergraduate thesis in mathematics, Carthage College, 2013.

[4] R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012.

[5] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[6] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, May 2003.

[7] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular Expression Types for XML. In *ICFP '00*, 2000.

[8] L. Nistor, D. Kurilova, S. Balzer, B. Chung, A. Potanin, and J. Aldrich. Wyvern: A simple, typed, and pure object-oriented language. In *Proceedings of the 5th Workshop on MechAnisms for SPEcialization, Generalization and inHerItance*, MASPEGHI '13, pages 9–16, New York, NY, USA, 2013. ACM.

[9] C. Omar, D. Kurilova, L. Nistor, B. Chung, A. Potanin, and J. Aldrich. Safely composable type-specific languages. In R. Jones, editor, *ECOOP 2014 âĂŞ Object-Oriented Programming*, volume 8586 of *Lecture Notes in Computer Science*, pages 105–130. Springer Berlin Heidelberg, 2014.

[10] OWASP. Open web application security project top 10.

[11] M. Sulzmann and K. Lu. Xhaskell – adding regular expression types to haskell. In O. Chitil, Z. HorvÃąth, and V. ZsÃşk, editors, *Implementation and Application of Functional Languages*, volume 5083 of *Lecture Notes in Computer Science*, pages 75–92. Springer Berlin Heidelberg, 2008.