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 generate strings containing commands that are executed by external systems like web browsers and database engines. If these strings are constructed from user input that has not been properly sanitized, this can lead to a variety of command injection vulnerabilities.

In this paper, we introduce *regular string types*, which classify strings known statically to be in a specified regular language and support operations like concatenation, substitution and coercion. Regular string types can be used to implement, in essentially a conventional manner, the parts of a web application or application framework that construct such command strings. Straightforward 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 type system like this should not require adopting a new programming language. Instead, we favor extensible type systems: new static type systems 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

Improper input sanitation is a leading cause of security vulnerabilities in web applications [OWASP]. Command injection attacks exploit improper input sanitation by inserting malicious code into an otherwise benign command. Modern web frameworks, libraries, and database abstraction layers attempt to ensure proper sanitation of user input. When these methods are unavailable or insufficient, developers implement custom sanitation techniques. Im both cases, sanitation algorithms are implemented using the language's regular expression capabilities and usually *replace* potentially unsafe strings with equivalent escaped strings.

In this paper, we present a type system for implementing and statically checking input sanitation techniques. Our solution suggests a more general approach to the integration of security concerns into programming language design. This approach is characterized by *composable* type system extensions which *complement* existing and well-understood solutions with compile-time checks.

To demonstrate this approach, we present a simply typed lambda calculus with *constrained strings*; that is, a set of string types parameterized by regular expressions. If s : $\mathsf{stringin}[r]$, then s is a string matching the language r. Additionally, we include an operation $\mathsf{rreplace}[r](s_1, s_2)$ which corresponds to the replace mechanism available in most regular expression libraries; that is, any substring of s_1 matching r is replaced with s_2. The type of this expression is the computed, and is likely "smaller" or more constrained than the type of s_1. Libraries, frameworks or functions which construct and execute commands containing input can specify a safe subset $\mathsf{stringin}[r_{\mathsf{spec}}]$ of strings, and input sanitation algorithms can construct such a string using $\mathsf{rreplace}$ or, optionally, by coercion (in which case a runtime check is inserted). We also show how this simple can be translated into a host language containing a regular expression library such that the safety guarantee of the extended language is preserved.

Summarily, we present a simple type system extension which ensures the absence of input sanitation vulnerabilities by statically checking input sanitation algorithms which use an underlying regular expression library. This approach is *composable* in the sense that it is a conservative extension. This approach is also *complementary* to existing input sanitation techniques which use string replacement for input sanitation.

1.1 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 lanugage 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 ensurees 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 [Jif][Ur/Web]. 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.

We are also unaware of any extensible programming languages which emphasize applications to security concerns (TRUE?).

Incorporating regular expressions into the type system is not novel. The XDuce system [**?**] typechecks XML schemas using regular expressions. We differ from this and related work in at least two ways. First, our system is defined within an extensible type system; second, and more importantly, we have demonstrated that regular expression types are applicable to the web security domain.

In conclusion, our system is novel in at least two ways:

- The safety guarantees provided by libraries and frameworks in popular languages are not as (statically) justified as is often belived (or even claimed).
- Our extension is the first major demonstration of how an extensible type system may be used to provide lightweight, composable security analyses based upon idiomatic code.

2. A TYPE SYSTEM FOR STRING SANITATION

The λ_S language is characterized by a type of strings indexed by regular expressions, together with operations on such strings which correspond to common input sanitation patterns. This section presents the grammar, typing rules and operational semantics for λ_S as well as an underlying language λ_P.

The system λ_S is the simply typed lambda calculus extended with *regular expression types*, which are string types ensuring a string belongs to a specified language. For instance, $S : \mathsf{stringin}[r]$ reads "s is a string matching r". the system includes an operation for replacing all instances of a pattern r in a string s_1 with another string s_2. Input sanitation algorithms – as implemented by developers or within popular libraries and frameworks – are often implemented in terms of this replace operation.

The language λ_P is a simple functional language extended with a minimal regular expression library. Any general purpose programming language could stand in for λ_P; for instance, SML or Python. In an implementation, our correctness results are modulo the underlying language's correct implementation of regular expression matching (see P-E-Replace).

$$\mathrm{r} \ ::= \ \epsilon \mid . \mid a \mid r \cdot r \mid r + r \mid r* \qquad a \in \Sigma$$

Figure 1: Regular expressions over the alphabet Σ.

Finally, we define a translation from our type system λ_S into λ_P which preserves the safety guarnatee codified in the string types of λ_S. Because our extension is conservative and its guarnatees are preserved under translation to an underlying language, it is highly composable with other analyses.

Unfortunately, we are unable to present full proofs here due to space constraints.

$$\begin{array}{lll}
\psi & ::= & \ldots \qquad \text{source types} \\
& \mid & \mathsf{stringin}[r]
\end{array}$$

$$\begin{array}{lll}
\mathrm{S} & ::= & \ldots \qquad \text{source terms} \\
& \mid & \mathsf{rstr}[s] \qquad s \in \Sigma^* \\
& \mid & \mathsf{rconcat}(S, S) \\
& \mid & \mathsf{rreplace}[r](S, S) \\
& \mid & \mathsf{rcoerce}[r](S)
\end{array}$$

Figure 2: Syntax for the string sanitation fragment of our source language, λ_S.

$$\begin{array}{lll}
\theta & ::= & \ldots \qquad \text{target types} \\
& \mid & \mathsf{string} \\
& \mid & \mathsf{regex}
\end{array}$$

$$\begin{array}{lll}
\mathrm{P} & ::= & \ldots \qquad \text{target terms} \\
& \mid & \mathsf{str}[s] \\
& \mid & \mathsf{rx}[r] \\
& \mid & \mathsf{concat}(P, P) \\
& \mid & \mathsf{preplace}(P, P, P) \\
& \mid & \mathsf{check}(P, P)
\end{array}$$

Figure 3: Syntax for the fragment of our target language, λ_P, containing strings and statically constructed regular expressions.

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

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

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

Figure 8: Translation from source terms (S) to target terms (P). The translation is type-directed in the Tr-Coerce cases.

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

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

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

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

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

Figure 4: Typing rules for our fragment of λ_S. The typing context Ψ is standard.

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

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

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

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

Figure 5: Big step semantics for our fragment of λ_S. Error propagation rules are omitted.

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

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

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

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

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

Figure 6: Typing rules for our fragment of λ_P. The typing context Θ is standard.

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

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

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

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

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

Figure 7: Big step semantics for our fragment of λ_P. Error propagation rules are omitted.

2.1 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 expression r. This distinction between the 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 well-known 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 expressions r, either $s \in \mathcal{L}\{r\}$ or $s \notin \mathcal{L}\{r\}$. Regular languages are closed under complements and concatenation. The regular expressions correspond bijectively to the regular languages.*

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

Definition 3 (`lreplace`)**.** The function $\texttt{lreplace}(r, r_1, r_2)$ produces a regular expression in which any sublanguage $\mathcal{L}\{r_1'\}$ of $\mathcal{L}\{r_1\}$ satisfying the condition $\mathcal{L}\{r_1'\} \subseteq \mathcal{L}\{r\}$ is replaced with $\mathcal{L}\{r_2\}$.

Lemma 4. *Closure and Totality of Replacement. If r, r_1 and r_2 are regular expressions, then $\texttt{lreplace}(r, r_1, r_2)$ is also a regular expression.*

Proof. By induction on r and closure properties of regular expressions. □

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

Proof. The proof proceeds by structural induction on r. The base case ($r = \alpha, \alpha \in \Sigma$) is trivial. The choice and sequential cases require lemma 1, and prove a more general property that $\texttt{lsubst}(r, s_1, s_2) \in \mathcal{L}\{r'\}$ for some r' isomorphic to $\texttt{lreplace}(r, r_1, r_2)$. The closure case follows from the induction hypothesis applied to any n-unwinding.

□

2.2 Safety of the Source and Target Languages

Lemma 6. *If $\Psi \vdash S : \mathsf{stringin}[r]$ 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 $\Theta \vdash P : \mathsf{regex}$ then $P \Downarrow \mathsf{rx}[r]$ such that r is a well-formed regular expression.*

We now prove safety for the string fragment of the source and target languages.

Theorem 8. *Safety for the String Fragment of P. Let S be a term in the source language. If $\Psi \vdash S : \mathsf{stringin}[r]$ then $S \Downarrow \mathsf{rstr}[s]$ and $\mathsf{rstr}[s] : \mathsf{stringin}[r]$, or else S err.*

```
1  @fn
2  def sanitize(s : string_in[r'.*']):
3    return (s.replace(r'"', '&quot;') # TODO: is this right?
4             .replace(r'<', '&lt;')
5             .replace(r'>', '&gt;'))
6
7  @fn
8  def results_query(s : string_in[r'[^"]*']):
9    return 'SELECT * FROM users WHERE name="' + s + '"'
10
11 @fn
12 def results_div(s : string_in[r'[^<>]*']):
13   return '<div>Results for ' + s + '</div>'
14
15 def main(db):
16   input = sanitize(user_input())
17   results = db.execute(results_query(input))
18   return results_div(input) + format(results)
```

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

Proof. By induction on the derivation of $\Psi \vdash S : \psi$. The interesting case is S-T-Replace, which requires lemma 5. The Coercion lemma additionally requires the second property of lemma 1.

□

Theorem 9. *Let P be a term in the target language. If $\Theta \vdash P : \theta$ then $P \Downarrow P'$ and $\Theta \vdash P' : \theta$.*

2.3 Translation Correctness

We now present the main correctness result.

Theorem 10. *If $S : \mathsf{rstr}[r]$ then there exists a P such that $[\![s]\!] = P$ and either: (a) $P \Downarrow \mathsf{str}[s]$ and $S \Downarrow \mathsf{rstr}[s]$, and $s \in langr$; 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 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.

□

3. IMPLEMENTATION IN ATLANG

wwwww Here, we use the argument annotation syntax introduced in Python 3 (a similar syntax is available for Python 2.6+, not shown).

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-prserving transformation. Unlike other approaches, our solution complements existing, familiar solutions while providing a strong guarantee that traditiona library and framework-based approaches are implemented and utilized correctly.

Papers that needs to be cited in this section:

- Ur/Web OSDI paper
- Jif?
- OWASP

```
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.

- XDuce and related papers.
- src?
- Ace or Wyvern paper?
- hotsos?
- Haskell extension paper
- Maybe some popular FOSS libraries/frameworks that do input sanitation?