A Type System for String Sanitation Implemented Inside a Python

ABSTRACT

ABSTRACT HERE

1. INTRODUCTION

Improper input sanitation is a leading cause of security vulnerabilities in web applications [OWASP]. For this reason, modern web frameworks and libraries use various techniques to ensure proper sanitation of arbitrary user input. When these methods are unavailable or insufficient, developers create ad hoc sanitation algorithms. Im most cases, sanitation algorithms – ad hoc or otherwise – are ultimately implemented using the language's regular expression capabilities. Therefore, a system capable of statically checking properties about operations performed using regular expressions will be expressive enough to capture real-world implementations of sanitation algorithms.

Input sanitation is the problem of ensuring that an arbitrary string is coerced into a safe form before potentially unsafe use. For example, preventing SQL injection attacks requires ensuring that any string coming from user input to a query does not contain unescaped SQL. The point at which arbitrary user input is concatenated into a SQL query is called a use site. Although we believe are general approach extends to a wider class of problems (e.g. sanitation algorithms for preventing XSS attacks might be definable using regular tree languages), these generalizations are beyond the scope of the present paper.

This paper presents a language extension, definable in the Ace programming language, for ensuring that input sanitation algorithms are implemented correctly with respect to use site specifications.

Instead of developing a top-down and holistic solution to the always-shifting security challenge, we propose an approach which allows developers to incorporate light-weight static analyses which best validate their privacy or security requirements. In this paper, we present a type system which ensures the absence of vulnerabilities and other bugs which arise from imporper input sanitation. We then demonstrate how this system is embeddable in an extensible type system.

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 defend the novelty and significance of our approach with respect to the state of the art in practice and in research.

Unlike frameworks provided by languages such as Haskell and Ruby, our type system provides a *static* guarantee that input is always properly sanitized before use. We achieve this by defining a typing relation which captures idiomatic sanitation algorithms. Type safety in our system relies upon several closure and decidability results about operations on regular languages which correspond to typical operations for sanitizing strings.

Libraries and frameworks available in functional programming language communities often make claims about security and sometimes even mention sophisticated type systems as evidence of freedom from injection-style attacks. In reality, many of these approaches ultimately depend upon the correct implementation of library or framework input sanitation functions. In these cases, our type system provides a stronger (compile-time) guarantee and is additionally orthogonal because the correctness of the library or framework itself could be checked by implementing its sanitation functions in terms of our system.

A number of research languages provide static guarantees that a program is free of input sanitation vulnerabilities. Most rely on some form of information flow TODO-nrf citations. Our extension to Ace differs from these systems in the ways following:

- Our system is a light-weight solution to a single class of sanitation vulnerabilities (e.g. we do not address Cross-Site Scripting). We present our system not as a comprehensive solution to the web security problem, but rather as evidence that composable, light-weight and simple analyses can address security problems.
- Our system is defined as a library in terms of an extensible type system, as opposed to a stand-alone lan-

guage. Instead of introducing new technologies and methodologies for addressing security problems, we provide a light-weight static analysis *based upon existing approaches to the problem* which are already well-understood by developers. We believe extensibility is a viable and attractive alternative to highly specialized programming languages.

- Ace is implemented in Python and shares its grammar. Since Python is a popular programming language among web developers, the barrier between our research and adopted technologies is far lower than is e.g. Ur/Web's.

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 light-weight, composable security analyses based upon idiomatic code.

1.2 Outline

An outline of this paper follows:

- In §2, we define the type system which is embedded in Ace. We include a type safety proof for the string segment of this language and prove the correctness of a translation to an underlying langguage P. In our theory, P is a simply typed lambda calculus equipped with a minimal regular expression library; in an implementation, P stands in for Python or another underlying general-purpose programming language.
- In §3, we discuss our implemention of this translation as a type system extension within the Ace programming language.

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 2 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, $S : \mathsf{rstr}[r]$ 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. For instance, a developer might all potentially unsafe charapcters with excaped versions of the same character. Regular expression types are used both to specify input sanitation algorithms, and at use sites as specifications. Note that runtime error states (S err) are introduced by coercion, not by replacement.

The language λ_P 2 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 has a regular expression library. In an implementation, our correctness results are modulo the underlying language's correct implementation of regular expression matching (see P-E-Replace).

Finally, we define a translation from our type system λ_S into λ_P.

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

Figure 1: Regular expressions over the alphabet Σ.

$$\begin{array}{llll} \psi & ::= & \ldots & \text{source types} \\ & | & \mathsf{stringin}[r] & \\ & & & \\ \text{S} & ::= & \ldots & \text{source terms} \\ & | & \mathsf{rstr}[s] & s \in \Sigma^* \\ & | & \mathsf{rconcat}(S, S) & \\ & | & \mathsf{rreplace}[r](S, S) & \\ & | & \mathsf{rcoerce}[r](S) & \end{array}$$

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

$$\begin{array}{llll} \theta & ::= & \ldots & \text{target types} \\ & | & \mathsf{string} & \\ & | & \mathsf{regex} & \\ & & & \\ \text{P} & ::= & \ldots & \text{target terms} \\ & | & \mathsf{str}[s] & \\ & | & \mathsf{rx}[r] & \\ & | & \mathsf{concat}(P, P) & \\ & | & \mathsf{replace}(P, P, P) & \\ & | & \mathsf{check}(P, P) & \end{array}$$

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

$$\boxed{\llbracket S \rrbracket = P}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 proofs for languages S and P and our translation correctness result all depend on some properties of regular languages. The crucial property is a relationship between string substitution – which is available in any regular expression library – and regular language substitution, which is a corresponding operation on languages instead of strings 5. The decidability of language substitution is what enables static analysis of sanitation algorithms implemented in terms of string replacement

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:*

(1): *If $s_1 \in \mathcal{L}\{r_1\}$ and $s_2 \in \mathcal{L}\{r_2\}$ then $s_1 s_2 \in \mathcal{L}\{r_1 r_2\}$*

(2): *For all strings s and expressions r, either $s \in \mathcal{L}\{r\}$ or $s \notin \mathcal{L}\{r\}$.*

(3): *Regular languages are closed under complements and concatenation.*

(4): *The regular expressions correspond bijectively to the regular languages.*

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

Definition 3 (lreplace)**.** The function $\mathtt{lreplace}(r, r_1, r_2)$ 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 $\mathtt{lreplace}(r, r_1, r_2)$ 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 $\mathtt{lsubst}(r, s_1, s_2) \in \mathtt{lreplace}(r, r_1, r_2)$.*

Proof. The proof proceeds by structural induction on r.

case $r = \alpha$. If $s_1 = \alpha$ then $\alpha \in \mathcal{L}\{r_1\}$ by assumption. Therefore, $\mathtt{lsubst}(r, s_1, s_2) = \mathtt{lsubst}(\alpha, \alpha, s_2) = s_2$ and $\mathtt{lreplace}(r, r_1, r_2) = \mathtt{lreplace}(\alpha, r_1, r_2)$. Since $s_1 = \alpha$ and $\alpha \in \mathcal{L}\{r_1\}$, $r_1 \cong \alpha | r_1'$ for some r_1'. Therefore, $\mathtt{lreplace}(\alpha, r_1, r_2) \cong \mathtt{lreplace}(\alpha, \alpha | r_1', r_2)$ by Lemma X(1). Finally, $s_2 \in \mathcal{L}\{r_2\}$ which implies $s_1 \in \mathcal{L}\{r_2 | r'\}$. If $s_1 \neq \alpha$ the $\mathtt{lsubst}(r, s_1, s_2) = s_1$ and $\mathtt{lreplace}(\alpha, r_1, r_2) = r_1$.

case $r = a|b$. Note that $[a|b/s_1]s_2 = \mathtt{lsubst}(a, \mathtt{lsubst}(b, s_1, s_2), s_2)$ and $\mathtt{lsubst}(b, s_1, s_2) \in \mathtt{lreplace}(b, r_1, r_2)$ by induction. Therefore, $\mathtt{lsubst}(a|b, s_1, s_2) \in \mathtt{lreplace}(a, \mathtt{lreplace}(b, r_1, r_2)$ by induction and the definition of lreplace. Finally, applying definitions once more, $\mathtt{lsubst}(a|b, s_1, s_2) \in \mathtt{lreplace}(a|b, r_1, r_2)$.

case $r = ab$. By a similar argument to the disjunctive case.

case $r = a*$. By considering the once unwinding of $a*$, noting that s_1 and s_2 are finite.

□

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 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.*

Proof. By induction on the derivation of $\Psi \vdash S : \psi$. The interesting case is S-T-Replace, which requires Lemma C.

S-T-Stringin-I: If $S = \mathsf{rconcat}(S_1, S_2) : \mathsf{stringin}[r]$ then $S \Downarrow S$ by S-E-RStr, and $\Psi \vdash S : \psi$ by assumption.

S-T-Concat: Suppose $S = \mathsf{rconcat}(S_1, S_2) : \mathsf{stringin}[r_1 r_2]$. By inversion, $\Psi \vdash S_1 : \mathsf{stringin}[r_1]$ and $\Psi \vdash S_1 : \mathsf{stringin}[r_2]$. It follows by induction that either S_1 err, S_2 err, or $S_1 \Downarrow \mathsf{rstr}[s_1]$ and $S_2 \Downarrow \mathsf{rstr}[s_2]$ for some $s_1 \in \mathcal{L}\{r_1\}$ and $s_2 \in \mathcal{L}\{r_2\}$. In the latter case $S \Rightarrow \mathsf{rstr}[s_1 s_2]$ by S-E-Concat and $\Psi \vdash \mathsf{rstr}[s_1 s_2] : \mathsf{str}[r_1 r_2]$ by 1. In the former cases, S err.

S-T-Replace: Suppose $S = \mathsf{rreplace}[r](S_1, S_2)$ and $\Psi \vdash S : \mathsf{stringin}[r']$. By inversion $\Psi \vdash S_1 : \mathsf{stringin}[r_1]$ and $\Psi \vdash S_2 : \mathsf{stringin}[r_2]$ such that $\mathtt{lsubst}(r, r_1, r_2) = r'$. By induction, S_1 err, S_2 err or $S_1 \Downarrow \mathsf{rstr}[s_1]$ and $S_2 \Downarrow \mathsf{rstr}[s_2]$ such that In the latter case, we know $\mathtt{lreplace}(r, s_1, s_2) \in \mathcal{L}\{\mathtt{lsubst}(r, r_1, r_2)\}$ by Lemma C; therefore by S-E-Replace, $S \Downarrow \mathsf{rstr}[s]$ such that $s \in \mathcal{L}\{\mathtt{lsubst}(r, r_1, r_2)\} = \mathcal{L}\{r'\}$. So by S-T-String-I, $\mathsf{rstr}[s] : \mathsf{stringin}[r']$. In the former cases, S err.

S-T-Coerce: Suppose $S = \mathsf{rcoerce}[r](S_1)$ and $S : \mathsf{stringin}[r]$. By inversion, $\Psi \vdash S_1 : \mathsf{stringin}[r']$. By induction, S_1 err or $S_1 \Downarrow \mathsf{rstr}[s]$. In the former case S err by propagation rules. In the latter case we have by property 2 of 1 that $s \in \mathcal{L}\{r\}$ or else $s \notin \mathcal{L}\{r\}$. If $s \in \mathcal{L}\{r\}$ then $\mathsf{rstr}[s] : \mathsf{stringin}[r]$. If $s \notin \mathcal{L}\{r\}$ then S err.

□

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

Proof. The proof proceeds by induction on the typing relation and is trivial give and inversion lemma for the typing relation. **We can write up this proof if we end up having enough space...** □

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 $s \in langr$.*

(b) *P err and S err.*

Proof. The proof proceeds by induction on the typing relation for S. Throughout the proof, properties from the closure lemma for regular languages are necessary; for brevity, we elide these references.

S-T-String-I: Let $S = \mathsf{rstr}[s]$ and suppose $\Psi \vdash \mathsf{rstr}[s] : \mathsf{stringin}[r]$. Choose $T = \mathsf{string}s$ and note that $[\![S]\!] = P$ by Tr-String. By P-E-String, $P \Downarrow \mathsf{string}s$ and by S-E-String $S \Downarrow \mathsf{rstr}[s]$. Finally, by inversion of S-T-Stringin-I, $s \in \mathcal{L}\{r\}$.

S-T-Concat: Let $S = \mathsf{rconcat}(S_1, S_2)$ and suppose $\Psi \vdash S : \mathsf{stringin}[r_1 r_2]$. By inversion, $\Psi \vdash S_1 : \mathsf{stringin}[r_1]$. It follows by induction that there exists a P_1 such that $[\![S_1]\!] = P_1$. By a similar argument for S_2 and r_2, there exists a P_2 such that $[\![S_2]\!] = P_2$. Choose $P = \mathsf{concat}(P_1, P_2)$.

We first prove property (a). Note that S_1 and P_1 are well typed (nrf ACTUALLY WE DON'T KNOW THAT P_1 IS WELL-TYPED!) and do not result in errors. Therefore, $S_1 \Downarrow \mathsf{rstr}[s_1]$ and $P_1 \Downarrow \mathsf{string}s_1$ for some $s_1 \in \mathcal{L}\{r_1\}$ by theorems 8 and 9 respectively. Similarly, $S_2 \Downarrow \mathsf{rstr}[s_2]$ and $P_2 \Downarrow \mathsf{string}s_2$ for some $s_2 \in \mathcal{L}\{r_2\}$. Therefore, $S \Downarrow \mathsf{rstr}[s_1 s_2]$ by S-E-Concat and $\mathsf{concat}(P_1, P_2) \Downarrow \mathsf{string}s_1 s_2$ by P-E-Concat. Finally, $s_1 s_2 \in \mathcal{L}\{r_1\} r_2$ by 1.

Consider property (b). If S_1 err then P_1 err by induction, and it follows that S err and P err by respective error propagation rules. Similarly, if S_2 err then P_2 err and it follow that S err and P err by induction and propagation.

S-T-Replace: Let $S = \mathsf{rreplace}[r](S_1, S_2)$ and suppose $\Psi \vdash S : \mathsf{stringin}[r']$ for some s. By inversion of S-T-Replace, , $\Psi S_1 : \mathsf{stringin}[r_1]$ and $\Psi : \mathsf{stringin}[r_2]$ such that $\texttt{lsubst}(r, r_1, r_2) = r'$. By induction, there exists some P_1, P_2 such that $[\![S_1]\!] = P_1$, $[\![S_2]\!] = P_2$ and either (a) or (b) holds.

If (a) holds then $S_1 \Downarrow \mathsf{rstr}[s_1]$ and $P_1 \Downarrow \mathsf{string}s_1$ for some $s_1 \in \mathcal{L}\{r_1\}$, and similarly for S_2, P_2 and some $s_2 \in \mathcal{L}\{r_2\}$. Therefore, by S-E-Replace, $S \Downarrow \mathsf{rstr}[s]$ for some $s = \texttt{lreplace}(r, s_1, s_2)$. Choose $P = \mathsf{preplace}(r, s_1, s_2)$. By a similar argument and P-E-Replace, $P \Downarrow \mathsf{string}s$ for some $s = \texttt{lreplace}(r, s_1, s_2)$. What remains to be shown is $\texttt{lreplace}(r, s_1, s_2) \in \mathcal{L}\{\texttt{lsubst}(r, r_1, r_2)\}$, which follows from Leamm D since $s_1 \in r_1$ and $s_2 \in r_2$.

If (b) holds for S_1 and P_1, then S err and P err by propagation rules. Similarly, if (b) hols for S_2 and P_2 then S err and P err by propagation rules.

S-T-Coerce: Let $S = \mathsf{rcoerce}[r](S')$ and suppose $\Psi \vdash \mathsf{rcoerce}[r](S) : \mathsf{stringin}[r]$. By inverstion $\Psi \vdash S' : \mathsf{stringin}[r']$ for an arbitrary r'. By induction there exists a P' such that $[\![S']\!] = P'$ and either (a) or (b) holds for S' and P'.

If (a) holds then $S' \Downarrow \mathsf{rstr}[s']$ and $P' \Downarrow \mathsf{string}s'$ for some $s' \in \mathcal{L}\{r'\}$. Note that either $s' \in \mathcal{L}\{r\}$ or $s' \notin \mathcal{L}\{r\}$ by property 2 of 1. Suppose $s' \in \mathcal{L}\{r\}$. Then $\mathsf{rcoerce}[r](S) \Downarrow \mathsf{rstr}[S']$ by S-E-Coerce. Choose $P = \mathsf{rx}\check{[}r]P'$ and note that $P \Downarrow \mathsf{string}s'$ by P-E-Coerce. Now suppose $s' \notin \mathcal{L}\{r\}$. Then S err and P err by P-E-Check-Err and S-E-Coerce-Err.

Finally, if (b) holds then S err and P err by propagation.

□

Papers that needs to be cited in this section:

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