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 that contain commands to be consumed by systems like web browsers and database engines. If these strings are constructed from user input that has not been properly sanitized, this can expose costly injection vulnerabilities.

In this paper, we introduce *regular string types*, which classify strings known statically to be in a specified regular language. The language of a string is tracked by the type system through operations like concatenation, substitution and coercion, so regular string types can be used to implement, in essentially a conventional manner, the parts of a web application or application framework that constructs commands. Simple type annotations at key points can be used to statically verify that sanitization has been performed correctly without introducing redundant run-time checks.

We take the position that to be practical, such a type system cannot require that programmers adopt a new programming language. Ideally, such a "special-purpose" type system would be distributed as a library that could safely be used together with other such type systems. We support this by specifying a sound translation to a simple language containing only strings and regular expressions, then discuss implementing the type system together with this translation as a library in `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. In 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 rreplace or, optionally, by coercion (in which case a runtime check is inserted). We also show how this system is 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;

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

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

In this section we define a language for statically checked string sanitation (λ_S) and prove that its correctness property is preserved under translation to a language with regular expression matching capabilities (λ_P). 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.
- 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 reesult about λ_S.

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

Figure 1: Regular expressions over the alphabet Σ.

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

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

$$\begin{array}{lcll}
\theta & ::= & \theta \to \theta & \text{target types} \\
 & | & \mathsf{string} & \\
 & | & \mathsf{regex} & \\
 & & & \\
\mathrm{P} & ::= & \lambda x.e & \text{target terms} \\
 & | & ee & \\
 & | & \mathsf{str}[s] & \\
 & | & \mathsf{rx}[r] & \\
 & | & \mathsf{concat}(P, P) & \\
 & | & \mathsf{preplace}(P, P, P) & \\
 & | & \mathsf{check}(P, P) &
\end{array}$$

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

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

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

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

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

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

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

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

$$\textsc{S-T-Replace}\ \frac{\begin{array}{c}\Psi \vdash S_1 : \mathsf{stringin}[r_1] \qquad \Psi \vdash S_2 : \mathsf{stringin}[r_2] \\ \mathtt{lreplace}(r, r_1, r_2) = r'\end{array}}{\Psi \vdash \mathsf{rreplace}[r](S_1, S_2) : \mathsf{stringin}[r']}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 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 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 $s \notin \mathcal{L}\{r\}$.*

Definition 2 (`lsubst`)**.** The replation $\texttt{lsubst}(r, s_1, s_2) = s$ 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 and reversal. □

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 refinitions of lsubst and lreplace; note that language substitutions over-approximate string substititons. □

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 and Sanitation Correctness for the String Fragment of P. Let S be a term in the source language. If $\Psi \vdash S : \mathsf{stringin}[r]$ then $S \Downarrow \mathsf{rstr}[s]$, $\Psi \mathsf{rstr}[s] : \mathsf{stringin}[r]$, and $s \in \mathcal{L}\{r\}$; 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, we proof a correctness result for λ_S which will be preserved under translation.

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

2.3 Translation Correctness

Theorem 10. *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. □

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

Proof. The theorem follows directly from canonical forms for translations and theorem 10. □

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 10, $P \Downarrow \mathsf{str}[s]$ implies that $S \Downarrow \mathsf{rstr}[s]$. By theorem 8, this together with the assumption that S is well-typed implies that $s \in \mathcal{L}\{r\}$. □

3. 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.

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?