A Type System for String Sanitation Implemented Inside a Python

Nathan Fulton Cyrus Omar Jonathan Aldrich

School of Computer Science
Carnegie Mellon University
{nfulton, comar, aldrich}@cs.cmu.edu

Abstract

ABSTRACT HERE

1. Introduction

In web applications and other settings, incorrect input sanitation often causes security vulnerabilities. In fact, ...OWASP says it's important... . Mention CVE stats. For this reason, modern web frameworks and libraries use various techniques to ensure proper sanitation of arbitrary user input. On the rare occasions when these methods are unavailable or insufficient, developers (hopefully) 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. If use site specifications are sufficient, then type checking ensures the absence of vulnerabilities and other bugs which arise from improper sanitation. Our extension focuses on sanitation algorithms which aim to prevent command-injection style attacks (e.g. SQL injection or RFI).

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 regular languages.

Libraries and frameworks available in functional programming language communities often make claims about security and sometimes even sophistically mention sophisticated type systems as evidence of freedom from injection-style attacked. However, even the specification that input is always sanitized properly before use is not actually captured by the type system or anywhere else; in fact, the full specification of these algorithms is rarely characterized by anything more specific than the type String $\rightarrow$ String, which is a class of safe input sanitation algorithms only in the most degenerate case.

A number of research languages provide actually static guarantees that a program is free of input sanitation vulnerabilities. Most rely on some form of information flow. TODO-nrf give citations and examples. 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 language. This is important for two reasons. First, our system is one of the first large examples written in Ace, and serves as an extended case study. Second, although our approach requires developers to adopt a new language, it does not require developers to adopt a language specifically for web development.
- 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.
- In general, our is a composable, light-weight and natural solution to a single problem – rather than a comprehensive solution to the web security problem. Our goal is to demonstrate that extensible type systems can capture static properties of common idioms, and thereby ensure safety without introducing much additional complexity.

Finally, 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; this first difference introduced an interesting class of design problems (see §???). TODO-nrf this is the second time this is used as a warrant. Factor out this argument and place it at the top??? Second, our focus on security required novel design decisions; for instance, the filter elimination form is unique to Ace.

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:

TODO-nrf real outline!

- In §2, we define the type system's static and dynamic semantics.
- Section 3 recalls some classical results about regular expressions and presents meta-theory for our system, including the main soundness theorem for λ_{CS}.
- Finally, §4 discusses our implemention of λ_{CS} as a type system extension within the Ace programming language.

2. A Type System for String Sanitation

The λ_{CS} 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 and semantics of λ_{CS}. The semantics are defined in terms of an internal language with at least strings and a regex filter function. These constraints are captured by the internal term valuations. The internal language does not necessarily need a regex filter function because any dynamic conversion is easily definable using a combination of filters and safe casting.

Theorem 1. *If* $S : \texttt{si\_str}[r]$ *then there exists a* T *such that* $[\![s]\!] = T$ *and either:*

- *(a)* $T \Downarrow \texttt{tstr}[s]$ *and* $S \Downarrow \texttt{si\_str}[s]$*, and* $s \in langr$*.*
- *(b)* T `err` *and* S `err`*.*

Proof. By induction on the derivation of $[\![S]\!] = T$. TODO-nrf add case magic.

- Tr-string. By assumption $[\![S]\!] = \texttt{tstr}[s]$. The (a) property follows. By S-E-Str, $S \Downarrow \texttt{strings}$ and by T-E-Str, $\texttt{tstr}[s] \Downarrow \texttt{tstr}[s]$. By source language type safety, it follows that $s \in \mathcal{L}\{r\}$.
- Tr-Concat. By inversion of the typing relaiton for the source lnaguage, $S_1 : \texttt{si\_str}[r_1]$ and $S_2 : \texttt{string\_in}[S_2]$. By induction on the left premise, $T_1 \Downarrow \texttt{tstr}[s_1]$, $S_1 \Downarrow \texttt{strings}_1$ and $s_1 \in \mathcal{L}\{r_1\}$ or else $T_1 err$. Similarly for T_2 and S_2. In the error cases, the propagations rules corresponding to S-E-Concat and T-E-Concat imply that $\texttt{tconcat}(T_1, T_2)$ `err` and $\texttt{siconcat}(S_1, S_2)$ `err`. In the non-error case, $\texttt{siconcat}(S_1, S_2) \Downarrow \texttt{si\_str}[s_1 \cdot s_2]$ by S-E-Concat and $\texttt{tconcat}(T_1, T_2) \Downarrow \texttt{tstr}[s_1 \cdot s_2]$ by T-E-Concat. Finally, $s_1 \in \mathcal{L}\{r_1\}$ and $s_2 \in \mathcal{L}\{r_2\}$ implies that $s_1 \cdot s_2 \in \mathcal{L}\{r\}$ by **??**.
- T-subst. Suppose $\texttt{subst}[r](S_1, S_2) : \texttt{si\_str}[s]$ for some x. By inverison, $S_1 : \texttt{si\_str}[r_1]$ and $S_2 : \texttt{si\_str}[r_2]$. Note that $s : [r/S_1]S_2 : \texttt{string\_in}[\texttt{tsubst}(r, r_1, r_2)]$ by S-T-Subst. By induction, $[\![S_1]\!] = T_1$ and $[\![S_2]\!] = T_2$. Either (a) or (b) holds for each.
  If (a) holds for both, the $S_1 \Downarrow \texttt{si\_str}[S_1]$ and $S_2 \Downarrow \texttt{si\_str}[S_2]$ for $s_1 \in \mathcal{L}\{r_1\}$ and $s_2 \in \mathcal{L}\{r_2\}$. Therefore, $\texttt{subst}[r](S_1, S_2) \Downarrow \texttt{si\_str}[\texttt{dosubst}(r, s_1, s_2,],)$. The same fact can be shown in an analogous manner for T_1 and T_2. Finally, $s_1 \in \mathcal{L}\{r_1\}$ and $s_2 \in \mathcal{L}\{r_2\}$, which implies that $\texttt{dosubst}(r, s_1, s_2) \in \mathcal{L}\{\texttt{subst}[r](s_1, s_2)\}$ by **??**.
  If (b) holds for either the left or right premise, then $\texttt{subst}[r](S_1, S_2)$ `err` by the propagation rules corresponding to S-T-subst, and $\texttt{tsubst}(S_1, S_2,$ `err`$)$ by the propagation rules corresponding to T-T-Subst.

□

$$
\begin{array}{lll}
\text{r} & ::= \epsilon \mid . \mid a \mid r \cdot r \mid r + r \mid r* & \text{Regex } (a \in \Sigma). \\
\psi & ::= & \text{Source language types} \\
 & \quad \texttt{string\_in}[r] & \text{Regular expression types} \\
 & \mid & \\
\text{S} & ::= & \text{Source language expressions} \\
 & \quad \texttt{si\_str}[S] & \\
 & \mid \texttt{siconcat}(S, S) & \text{String concatenation} \\
 & \mid \texttt{subst}[r](S, S) & \\
 & \mid \texttt{coerce}[r](S) & \\
\theta & ::= & \text{Target language types} \\
 & \quad \texttt{string} & \\
 & \mid \texttt{rx}[] & \\
\text{P} & ::= & \text{Internal expressions} \\
 & \quad \texttt{tstr}[P] & \\
 & \mid \texttt{siconcat}(P, P) & \\
 & \mid \texttt{tsubst}(r, P, P) & \\
 & \mid \texttt{tcheck}(r, P) &
\end{array}
$$

Figure 1. Syntax for the string sanitation fragment of λ_{CS}

2.1 Properties of Regular Languages

The regular languages are the smallest set generated by regular expressions defined in Figure 1.

Theorem 2. *Closure Properties. The regular expressions are closed under complements and concatenation.*

Proof. See [**?**]. □

Theorem 3. *Coercion Theorem. Suppose that R and L are regular expressions, and that $s \in R$ is a finite string. Let $s' := coerce(R, L, s)$ with all maximal substrings recognized by L replaced with ϵ. Then s' is recognized by $(R \backslash L)+\emptyset$ and the construction of $R \backslash L$ is decidable.*

Proof. Let F, G be FAs corresponding to R and L, and let G' be G with its final states inverted (so that G' is the complement of L). Define an FA H as a DFA corresponding to the NFA found by combining F and G' such that H accepts only if R and L' accept or if s is empty (this construction may result in an exponential blowup in state size.) Clearly, H corresponds to $R \backslash L + \emptyset$. Thus, the construction of $R \backslash L + \emptyset$ is decidable.

If $R \subset L$, $s' = \emptyset$. If $L \subset R$, either $s' = \emptyset$, or $s' \in R$ and $s' \notin L$. If R and L are not subsets of one another, then it may be the case that L recognizes part of R. Consider L as the union of two languages, one which is a subset of R and one which is disjoint. The subset language is considered above and the disjoint language is inconsequential. □

$$\frac{s \in \mathcal{L}\{r\}}{\Gamma \vdash \texttt{si\_str}[s] : \texttt{string\_in}[r]} \quad \text{S-T-INCLUDE}$$

$$\frac{\Gamma \vdash S_1 : \texttt{string\_in}[r_1] \qquad \Gamma \vdash S_2 : \texttt{string\_in}[r_2]}{\Gamma \vdash \texttt{siconcat}(S_1, S_2) : \texttt{si\_str}[r_1 \cdot r_2]} \quad \text{S-T-CONCAT}$$

$$\frac{\Gamma \vdash S_1 : \texttt{string\_in}[r_1] \qquad \Gamma \vdash S_2 : \texttt{string\_in}[r_2] \qquad \texttt{lsubst}(r, r_1, r_2) = r'}{\Gamma \vdash \texttt{subst}[r](S_1, S_2) : \texttt{string\_in}[r']} \quad \text{S-T-SUBST}$$

$$\frac{\Gamma \vdash S : \texttt{string\_in}[r]}{\Gamma \vdash \texttt{coerce}[r'](s) : \texttt{string\_in}[r']} \quad \text{S-T-COERCE}$$

Figure 2. Typing rules for source langauge terms (S). The metavariable s ranges over string literals.

$$\frac{\cdot}{\texttt{si\_str}[s] \Downarrow \texttt{si\_str}[s]} \quad \text{S-E-INCLUDE}$$

$$\frac{S_1 \Downarrow \texttt{si\_str}[s_1] \qquad S_2 \Downarrow \texttt{si\_str}[s_2]}{\texttt{siconcat}(S_1, S_2) \Downarrow \texttt{si\_str}[s_1 \cdot s_2]} \quad \text{S-E-CONCAT}$$

$$\frac{S_1 \Downarrow \texttt{si\_str}[s_1] \qquad S_2 \Downarrow \texttt{si\_str}[s_2] \qquad \texttt{dosubst}(r, s_1, s_2) = s}{\texttt{subst}[r](S_1, S_2) \Downarrow s} \quad \text{S-E-SUBST}$$

$$\frac{S \Downarrow \texttt{si\_str}[s] \qquad s \in \mathcal{L}\{r\}}{\texttt{coerce}[r](S) \Downarrow \texttt{si\_str}[s]} \quad \text{S-E-COERCE-OK}$$

$$\frac{\texttt{coerce}[r](S) \Downarrow \texttt{si\_str}[s] \qquad s \notin \mathcal{L}\{r\}}{S\ \texttt{err}} \quad \text{S-E-COERCE-NOTOK}$$

Figure 3. Big step semantics for the string fragment of the source language. There are also unmentioned error propagation rules for for S-E-Concat and S-E-Subst.

$$\frac{\cdot}{\Omega \vdash \texttt{tstr}[s] : \texttt{string}} \quad \text{P-T-STRING}$$

$$\frac{\Omega \vdash P_1 : \texttt{string} \qquad \Omega \vdash P_2 : \texttt{string}}{\Omega \vdash \texttt{tconcat}(P_1, P_2) : \texttt{string}} \quad \text{P-T-CONCAT}$$

$$\frac{\Omega \vdash P_1 : \texttt{string} \qquad \Omega \vdash P_2 : \texttt{string}}{\Omega \vdash \texttt{tsubst}(\texttt{rx}[r], P_1, P_2) : \texttt{string}} \quad \text{P-T-SUBST}$$

$$\frac{\Omega \vdash P : \texttt{string}}{\Omega \vdash \texttt{tcheck}(\texttt{rx}[r], s) : \texttt{string}} \quad \text{P-T-CHECK}$$

Figure 4. Typing rules for target langauge terms (P). The metavariable s ranges over string literals.

$$\frac{\cdot}{\texttt{tstr}[s] \Downarrow \texttt{tstr}[s]} \quad \text{P-E-STR}$$

$$\frac{\cdot}{\texttt{rx}[r] \Downarrow \texttt{rx}[r]} \quad \text{P-E-INCLUDE}$$

$$\frac{P_1 \Downarrow \texttt{tstr}[s_1] \qquad P_2 \Downarrow \texttt{tstr}[s_2]}{\texttt{tconcat}(P_1, P_2) \Downarrow \texttt{tstr}[s_1 \cdot s_2]} \quad \text{P-E-CONCAT}$$

$$\frac{P_1 \Downarrow \texttt{rx}[r] \qquad P_2 \Downarrow \texttt{tstr}[s_2] \qquad P_3 \Downarrow \texttt{tstr}[s_3] \qquad \texttt{dosubst}(r, s_2, s_3) = s}{\texttt{tsubst}(r, P_2, P_3) \Downarrow \texttt{tstr}[s]} \quad \text{P-E-SUBST}$$

$$\frac{P \Downarrow \texttt{si\_str}[s] \qquad s \in \mathcal{L}\{r\}}{\texttt{tcheck}(r, P) \Downarrow \texttt{tstr}[s]} \quad \text{P-E-CHECK-OK}$$

$$\frac{\texttt{tcheck}(r, P) \Downarrow \texttt{tstr}[s] \qquad s \notin \mathcal{L}\{r\}}{P\ \texttt{err}} \quad \text{P-E-CHECK-NOTOK}$$

Figure 5. Big step semantics for the target lanugage. There are also unimplemented error propagation rules for for Te-Concat and Te-Subst.

$$\frac{}{[\![\texttt{si\_str}[s]]\!] = \texttt{tstr}[s]} \quad \text{TR-STRING}$$

$$\frac{[\![S_1]\!] = P_1 \qquad [\![S_2]\!] = P_2}{[\![\texttt{siconcat}(S_1, S_2)]\!] = \texttt{tconcat}(P_1, P_2)} \quad \text{TR-CONCAT}$$

$$\frac{[\![S_1]\!] = P_1 \qquad [\![S_2]\!] = P_2}{[\![\texttt{subst}[r](S_1, S_2)]\!] = \texttt{tsubst}(\texttt{rx}[r], P_1, P_2)} \quad \text{TR-SUBST}$$

$$\frac{S : \texttt{si\_str}[r] \qquad \mathcal{L}\{r'\} \subseteq \mathcal{L}\{r\}}{[\![\texttt{coerce}[r'](S)]\!] = \texttt{tstr}[s]} \quad \text{TR-COERCE-OK}$$

$$\frac{[\![S]\!] = P \qquad S : \texttt{si\_str}[r] \qquad \mathcal{L}\{r'\} \not\subseteq \mathcal{L}\{r\}}{[\![\texttt{coerce}[r'](S)]\!] = \texttt{tcheck}(\texttt{rx}[r'], P)} \quad \text{TR-COERCE-NOTOK}$$

Figure 6. Translation from source terms (S) to target terms (P). The translation is type-directed in the Tr-Coerce cases.

```
tycon STRINGIN of R {                                              (1)
  iana {λopidx:String.λtyidx:R.λa:list[Arg].                       (2)
    arity0 a (check opidx tyidx ▷ (◁(opidx)))}                     (3)
  esyn {λopidx:1 + (R + R).λa:list[Arg].                           (4)
    case opidx of inl(_) ⇒                                         (5)
      arity2 a λa1:Arg.λa2:Arg.                                    (6)
        rsyn a1 λr1:R.λi1:ITm.                                     (7)
        rsyn a2 λr2:R.λi2:ITm.                                     (8)
          (STRINGIN⟨rseq(r1; r2)⟩,                                 (9)
           ▷ (iconcat(◁(i1); ◁(i2))))                              (10)
      | inr(opidx') ⇒ d}}                                          (11)
```

Figure 8. Definition of ...

$$
\begin{array}{lcl}
\textbf{concat} & := & \mathsf{inl}[1, R + R](b) \\
\textbf{replace} & := & \lambda \mathbf{r}{:}R.\mathsf{inr}[1, R + R](\mathsf{inl}[R, R](\mathbf{r})) \\
\textbf{coerce} & := & \lambda \mathbf{r}{:}R.\mathsf{inr}[1, R + R](\mathsf{inr}[R, R](\mathbf{r}))
\end{array}
$$

Figure 9. Definitions

Note that we never insert an internal check where the type of the string implies that a check must succeed. Furthermore, expensive calculations (such as language inclusion or `dsubst`, all occur at compile time. Since sanitation problems are generally expressible with smalle, simple expressions, we believe that the compile-time overhead is not significant enough to prohibiour target usecase. Informal experimentation with our implementation seems to support this assertion.

Due to Ei-Replace, all of our meta-theory depends upon a correct implementation of both regular expression replacement and inclusion checking in the internal language. We believe this assumption is okay for two reasons. First, our system is still superior to the status quo, which relies upon the correctness of these libraries *in addition to* correct application logic. Second, regular expression libraries are generally well-tested and there exist verified implementations.

2.2 Type Safety Proof

2.3 New Theorems

Theorem 4 (Progress for Internal Language)**.**

Theorem 5 (Preservation for Internal Language)**.**

Theorem 6 (Representational Consistency)**.** *If* $\cdot \vdash e : \tau \Downarrow i$ *then* $\cdot \vdash i : \sigma$ *and* $\sigma \equiv \mathtt{rep}(\tau)$.

Theorem 7 (Main Soundness Result)**.** *If* $\cdot \vdash e : \mathtt{si\_str}[r] \Downarrow i$ *then either* $i \Downarrow \mathtt{err}$ *or if* $i \Downarrow v$ *for* v*??? then* $v \in r$.

3. Implementation Inside a Python

3.1 Background: Ace

TODO: Make a TR out of the OOPSLA submission.

3.2 Explicit Conversions

3.3 Adding Subtyping to Ace

3.4 Theory

4. Related Work

5. Discussion

τ		$\lfloor\tau\rfloor_{\lambda_{\text{Ace}}}$	$\lfloor\tau\rfloor_{\mathsf{ace}}$
`si_str`$[r]$		STRINGIN$[r]$	`string_in["`r`"]`
e		$\lfloor e\rfloor_{\lambda_{\text{Ace}}}$	$\lfloor e\rfloor_{\mathsf{ace}}$
`str`$[r, s]$	synthetic position	intro[str$[s]$]() : STRINGIN[rx$[r]$]	`"`s`" (string_in["`r`"])`
	analytic position	intro[str$[s]$]()	`"`s`"`
`concat`$(e_1; e_2)$		$\lfloor e_1\rfloor_{\lambda_{\text{Ace}}} \cdot$ elim[**concat**]$(\lfloor e_2\rfloor_{\lambda_{\text{Ace}}})$	$\lfloor e_1\rfloor_{\mathsf{ace}}$ `+` $\lfloor e_2\rfloor_{\mathsf{ace}}$
`replace`$[r](e_1; e_2)$		$\lfloor e_1\rfloor_{\lambda_{\text{Ace}}} \cdot$ elim[**replace** rx$[r]$]$(\lfloor e_2\rfloor_{\lambda_{\text{Ace}}})$	$\lfloor e_1\rfloor_{\mathsf{ace}}$`.replace("`$r$`",` $\lfloor e_2\rfloor_{\mathsf{ace}}$`)`
`coerce`$[r](e)$		$\lfloor e\rfloor_{\lambda_{\text{Ace}}} \cdot$ elim[**coerce** rx$[r]$]()	$\lfloor e\rfloor_{\mathsf{ace}}$`.coerce("`$r$`")`

Figure 7. Translation of λ_{CS} to λ_{Ace} and Ace.