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 a type safety proof for λ_{CS} based upon these properties.
- Finally, §4 discusses our implemention of λ_{CS} as a type system extension within the Ace programming language.

$\langle r \rangle$ ::=	$\epsilon \mid . \mid a \mid r \cdot r \mid r + r \mid r*$	Regex ($a \in \Sigma$).
$\langle t \rangle$ ::=		terms:
	$\texttt{filter}(\texttt{string\_in}[r], t)$	filter substrings
|	$[\texttt{string\_in}[r]](t)$	safe conversion
|	$\texttt{dconvert}(\texttt{string\_in}[r], t)$	unsafe conversion
$\langle iv \rangle$ ::=		internal values:
	's'	internal string
|	$\texttt{ifilter}(r, \texttt{istring}(s))$	internal filter
$\langle v \rangle$ ::=		values:
	"s"	string
$\langle T \rangle$ ::=		types:
	$\texttt{string}$	Strings
|	$\texttt{string\_in}[r]$	Regular language strings
$\langle \Gamma \rangle$ ::=	$\emptyset \mid \Gamma, x : T$	typing context

Figure 1. Syntax of λ_{CS}

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 (`ival`). 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.

The λ_{CS} language gives static semantics for common regular expression library functions. In this treatment, we include concatenation and filtering. The `filter` function removes all instances of a regular expression in a string, while concatenation ($+$) concatenates two strings.

2.1 Typing

The $\texttt{string\_in}[r]$ type is parameterized by regular expressions; if $e : \texttt{string\_in}[r]$, then $e \in r$. Mapping from an arbitrary `string` to a $\texttt{string\_in}[r]$ requires defining an algorithm – in terms of filter – for converting a $\texttt{string\_in}[.*]$ into a $\texttt{string\_in}[r]$. The static semantics of the language defines the types of operations on regular expressions in terms of well-understood properties about regular lanugages; we recall these properties in section 3.

2.2 Dynamics

There are two evaluation judgements: $e : T \Rightarrow e'$ and $e : T \rightsquigarrow i$. The $\Rightarrow$ relation is between λ_{CS} expressions, while the $\rightsquigarrow$ relation is a mapping from λ_{CS} expressions into internal language expressions i such that i `ival`.

Safety of the evaluation relation depends upon an injective mapping from λ_{CS} types info internal language types. This relation, h, is defined below.

2.3 Type Safety

The type safety proof relies upon some assumptions about the type system and dynamics of the internal language, as well as some properties of regular languages.

There must exist a translation from λ_{CS} types to the types of the internal language. For the remainder of this paper, we call the type translation function h.

Definition 1 (Type Translation Function h)**.** The type translation function $h : Type \to IType$ is defined as follows:

- $\forall r.h(\texttt{string\_in}[r]) = \texttt{istring}$
- $h(\texttt{string}) = \texttt{istring}$

Additionally, we assume that the internal language contains an implementation of strings, together with operations for concatenation and filtering by regular expression.

Definition 2 (Types of internal values)**.** Let 's' range over string literals and r over regular expressions. Internal values are typed as follows:

- If $e =$'s' then e : `istring`.
- If $e = \texttt{ifilter}(r, \text{'}s\text{'})$ then e : `istring`.
- If $e =$ 's_1' + 's_2' then e : `istring`.

For simplicity, we assume a fixed translation from λ_{CS} regular expressions to regular expressions recognizable by the internal language's regex library (in practice, a fixed translation is acceptable.) To summarize, we assume an internal language containing a string type together with operations for string concatentation and filtering. We expect closure over strings for both operations. Finally, recall that `ifilter` is only needed for dynamic casts, which may be removed without descreasing the expressivity or even usability of the language. Finally, the semantics of the filter function are defined in terms of `rl_filter`, which is a static version of `ifilter`.

2.4 Properties of Regular Languages

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

Theorem 3. *Closure Properties. The regular expressions are closed under complements and concatenation.*

Proof. See [**?**]. □

Theorem 4. *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. □

$$\frac{}{\texttt{string} \equiv \texttt{string\_in}[.*]} \quad \text{T-EQUIV-TOP}$$

$$\frac{\mathcal{L}\{r\} = \mathcal{L}\{r'\}}{\texttt{string\_in}[r] \equiv \texttt{string\_in}[r']} \quad \text{T-EQUIV-STRING\_IN}$$

$$\frac{e \in \mathcal{L}\{r\}}{\Gamma \vdash e : \texttt{string\_in}[r]} \quad \text{T-EQUIV-INCLUDE}$$

$$\frac{\Gamma \vdash e : \texttt{string}}{\Gamma \vdash e : \texttt{string\_in}[e]} \quad \text{T-MINIMUM}$$

$$\frac{\Gamma \vdash e_1 : \texttt{string\_in}[r_1] \qquad \Gamma \vdash e_2 : \texttt{string\_in}[r_2]}{\Gamma \vdash e_1 + e_2 : \texttt{string\_in}[r_1 \cdot r_2]} \quad \text{T-CONCAT}$$

$$\frac{\Gamma \vdash e : \texttt{string\_in}[r] \qquad \mathcal{L}\{r'\} \subseteq \mathcal{L}\{r\}}{\Gamma \vdash [\texttt{string\_in}[r']](e) : \texttt{string\_in}[r']} \quad \text{T-CONVERT}$$

$$\frac{\Gamma \vdash e : \texttt{string\_in}[r]}{\Gamma \vdash \texttt{dconvert}(\texttt{string\_in}[r'], e) : \texttt{string\_in}[r']} \quad \text{T-DCONVERT (OPTIONAL)}$$

$$\frac{\Gamma \vdash t : \texttt{string\_in}[r]}{\Gamma \vdash \texttt{filter}(\texttt{string\_in}[r'], t) : \texttt{string\_in}[(r \backslash r') + \emptyset]} \quad \text{T-FILTER}$$

Figure 2. Typing relation for λ_{CS}

$$\frac{}{\text{“}s\text{”} : \texttt{string\_in}[r]\ \texttt{val}} \qquad \text{E-STRINVAL}$$

$$\frac{}{\text{‘}e\text{’} : \texttt{istring}\ \texttt{ival}} \qquad \text{E-IVAL}$$

$$\frac{e\,\texttt{ival}}{\texttt{ifilter}(r, e)\ \texttt{ival}} \qquad \text{E-IFILTERVAL}$$

$$\frac{e\ \texttt{val}}{e : \texttt{string} \rightsquigarrow \text{‘}e\text{’}} \qquad \text{E-STRING}$$

$$\frac{e\ \texttt{val}}{e : \texttt{string\_in}[r] \rightsquigarrow \text{‘}e\text{’}} \qquad \text{E-STRING\_IN}$$

$$\frac{e_1\ \texttt{val} \qquad e_2\ \texttt{val}}{e_1 + e_2 : \texttt{string\_in}[r_1 + r_2] \rightsquigarrow \text{‘}e_1\text{’} + \text{‘}e_2\text{’}} \qquad \text{E-CONCATVAL}$$

$$\frac{e_1 : \texttt{string\_in}[r_1] \Rightarrow {e_1}'}{e_1 + e_2 : \texttt{string\_in}[r_1 + r_2] \Rightarrow {e_1}' + e_2} \qquad \text{E-CONCATL}$$

$$\frac{e_2 : \texttt{string\_in}[r_2] \Rightarrow {e_2}'}{e_1 + e_2 : \texttt{string\_in}[r_1 + r_2] \Rightarrow e_1 + {e_2}'} \qquad \text{E-CONCATR}$$

$$\frac{e\ \texttt{val}}{\texttt{filter}(\texttt{string\_in}[r'], e) : \texttt{string\_in}[r \backslash r' + \emptyset] \Rightarrow \texttt{rl\_filter}(r, e)} \qquad \text{E-FILTERVAL}$$

$$\frac{e : \texttt{string\_in}[r] \Rightarrow e'}{\texttt{filter}(\texttt{string\_in}[r'], e) : \texttt{string\_in}[r \backslash r' + \emptyset] \Rightarrow \texttt{filter}(\texttt{string\_in}[r], e')} \qquad \text{E-FILTER}$$

$$\frac{e\ \texttt{val}}{[\texttt{string\_in}[r']](e) : \texttt{string\_in}[r'] \rightsquigarrow \text{‘}e\text{’}} \qquad \text{E-CONVERTVAL}$$

$$\frac{e : \texttt{string\_in}[r] \Rightarrow e'}{[\texttt{string\_in}[r']](e) : \texttt{string\_in}[r'] \Rightarrow [\texttt{string\_in}[r']](e')} \qquad \text{E-CONVERT}$$

$$\frac{e : \texttt{string\_in}[r] \Rightarrow e'}{\texttt{dconvert}(\texttt{string\_in}[r'], e) : \texttt{string\_in}[r'] \Rightarrow \texttt{dconvert}(\texttt{string\_in}[r'], e')} \qquad \text{E-DCONVERTVAL (OPTIONAL)}$$

$$\frac{e\ \texttt{val}}{\texttt{dconvert}(\texttt{string\_in}[r'], e) : \texttt{string\_in}[r'] \rightsquigarrow \texttt{ifilter}(r', \text{‘}e\text{’})} \qquad \text{E-DCONVERT (OPTIONAL)}$$

Figure 3. SOS rules for λ_{CS}

2.5 Type Safety Proof

Theorem 5 (Preservation)**.** *Let T be a type in λ_{CS} and $h(T) = \sigma$ the corresponding type in the internal language. For all terms e:*

- *If $e : T$ and $e : T \rightsquigarrow i$ then $i : \sigma$ such that $h(T) = \sigma$.*
- *If $e : T$ and $e : T \Rightarrow e'$ then $e' : T$.*

Proof. The proof is a straightforward induction on the derivation of the combined evaluation relation.

E-Ival, E-Ifilterval. Both cases hold since the terms at hand are not λ_{CS} terms.

E-Stringval, E-strval. Both cases hold since no reduction is possible.

E-String. By the definition of typing for internal terms, 'e' : `istring`. It suffices to show that $h(\texttt{string}) = \texttt{istring}$, which follows from the definition of h.

E-String_in. By the definition of typing for internal terms, 'e' : `istring`. It suffices to show that $h(\texttt{string\_in}[r]) = \texttt{istring}$, which follows from the definition of h for arbitrary r.

E-concatval. By the definition of typing for internal terms, 'e_1' + 'e_2' : `istring`. So it suffices to show that $h(\texttt{string\_in}[r_1 + r_2]) = \texttt{istring}$, which follows from the definition of h for arbitrary r_1, r_2.

E-concatR, E-concatL. Consider E-concatR. By induction, e_1' : `string_in`l_1. By inversion of T-Concat at the premise, e_2 : `string_in`r_2. Therefore, $e_1 + e_2$: $\texttt{string\_in}[r_1 + r_2]$. The left rule is symmetric.

E-Filterval. We have that $\texttt{filter}(\texttt{string\_in}[r'], e)$: $\texttt{string\_in}[r \backslash r' + \emptyset]$. By inversion of T-Filter, e : $\texttt{string\_in}[r]$. By T-Equiv-string_in (which is bidirectional), $e \in \mathcal{L}\{r\}$. By the Coercion Theorem, $\texttt{rl\_filter}(r, e) \in \mathcal{L}\{r \backslash r' + \emptyset\}$. By T-Equiv-string_in, $e \in \mathcal{L}\{r\}$ and $\texttt{rl\_filter}(r, e) \in \mathcal{L}\{r \backslash r' + \emptyset\}$ implies $\texttt{rl\_filter}(r, e)$: $\texttt{string\_in}[r \backslash r' + \emptyset]$.

E-Filter. By inversion, e : $\texttt{string\_in}[r'] \Rightarrow e'$ so by the induction e' : $\texttt{string\_in}[r']$. Therefore, $\texttt{filter}(\texttt{string\_in}[r], e')$: $\texttt{string\_in}[r \backslash r' + \emptyset]$ by T-Filter.

E-Convertval. It suffices to show that $h(\texttt{string\_in}[r]) = \texttt{istring}$, which is true by definition.

E-Convert. By inversion and induction, e' : $\texttt{string\_in}[r]$. We know that $[\texttt{string\_in}[r']](e)$: $\texttt{string\_in}[r']$, so by inversion of T-Convert $\mathcal{L}\{r'\} \subseteq \mathcal{L}\{r\}$. It follows that $[\texttt{string\_in}[r']](e')$: $\texttt{string\_in}[r']$.

E-DConvert. By inversion, e : $\texttt{string\_in}[r'] \Rightarrow e'$. By the induction hypothesis, e' : $\texttt{string\_in}[r']$; therefore, by T-Dconvert, $\texttt{dconvert}([, \texttt{string\_in})[r]](e)$: $\texttt{string\_in}[r]$.

E-DConvertval. By the definition of typing for internal terms, $\texttt{ifilter}((, r), \text{'}e\text{'}) : \texttt{istring}$. It suffices to show that $h(\texttt{string\_in}[r_1 + r_2]) = \texttt{istring}$, which follows from the definition of h.

□

Theorem 6 (Progress)**.** *If $e : T$ then $e : T \Rightarrow^* e' \rightsquigarrow^* i$ where i* `ival`.

Proof. By induction on the derivation of $e : T$. For **T-Equiv-Include**, note that $e \in \mathcal{L}\{r\}$, $e =$ "s" for some s; therefore, e `val` by E-Strinval. We have that e `val` and e : $\texttt{string\_in}[r]$, so $e \rightsquigarrow$ 'e' by E-string_in. The remaining cases follow by induction on the hypotheses and application of corresponding evaluation rules. □

3. Implementation Inside a Python

TODO-nrf rewrite The λ_{CS} language is implemented as a type system extension in Ace; this extension is illustrated in the examples at the beginning of this paper.

= Computing a regular expression representating the language $R \backslash S$ is necessary in order to type-check expressions in which the filter function occurs. This language is computed by translating R and S into finite automata, complementing the final states of S, then constructing the cross-product of R and S'. Type checking terminates only because this construction is decidable.

In addition to this construction, other more typical operations – such as equality checks for regular expressions and regular expression matching – are also necessary. For these reasons, the Ace extension implementing λ_{CS} includes a library implementing each of these constructions.

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