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 ultimately generate strings containing commands, which are executed by systems like web browsers and database engines. Strings constructed from user input that has not been properly sanitized can thus cause 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

Command injection vulnerabilities are among the most common and severe in modern web applications [7]. They arise because web applications, at their boundaries, must control external systems that expose string-based command interfaces. For example, web browsers are controlled using HTML and Javascript sent from a server as a string, and database engines execute SQL queries also sent as strings. When these commands contain data derived from user input, care must be taken to ensure that the user cannot provide an input that will subvert the intended command. For example, a SQL query constructed using string concatenation exposes a SQL injection vulnerability if `name` is controlled by a user:

```
'SELECT * FROM users WHERE name="' + name + '"'
```

If a malicious user enters the name `'"; DROP TABLE users --'`, the entire database could be erased.

To avoid this problem, the program must *sanitize* user input. For example, in this case, the developer (or, more often, a framework) might define a function `sanitize` that escapes any double quotes, which are necessary in order to force uesr input to be interpreted as anything other than a string literal.

We observe that most such sanitization techniques can be understood in terms of *regular languages*. For example, `name` should be a string in the language described by the regular expression `([^"\]|(\")|(\\))*` – a sequence consisting of characters other than quotation marks or backslashes, or escaped quotation marks or escaped backslashes. Concrete syntax like this can be understood to desugar, in a standard way, to the abstract syntax for regular expressions shown in Figure 1. We will work with this "core" for simplicity, and assume basic familiarity with regular expressions [4].

In this paper, we present a static type system that tracks the regular language a string belongs to. We take advantage of closure and decidability properties of regular languages to support a number of useful operations on values of such *regular string types*. These make it possible to implement sanitation protocols like the one just described in an essentially conventional manner. The result is a system where the fact that a string has been *correctly* sanitized becomes manifest in its type. Missing calls to sanitization functions are detected statically, and, importantly, so are *incorrectly implemented sanitization functions* (i.e. these functions need not be trusted). Run-time checks are only used when going from less precise to more precise types (e.g. at the edges of the system, where user input has not yet been validated) and no additional space overhead is required.

TODO: need to update remainder of intro ... 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 ex-

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.

ecute 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 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; 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 ensures 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 [1]. 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.

Incorporating regular expressions into the type system is not novel. The XDuce system [6, 5] checks XML documents against schemas using regular expressions. Similarly, XHaskell [8] focuses on XML documents. We differ from this and related work in at least three ways:

- Our system is defined within an extensible type system.
- We demonstrate that regular expression types are applicable to the web security domain, whereas previous work on regular expression types focused on XML schemas.
- Although our static replacement operation is definable in some languages with regular expression types, we are the first to expose this operation and connect the semantics of regular language replacement with the semantics of string substitution via a type safety and compilation correctness argument.

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

Figure 1: Regular expressions over the alphabet Σ.

In conclusion, our contribution is a type system, implemented within an extensible type system, for checking the correctness of input sanitation algorithms.

2. A TYPE SYSTEM FOR STRING SANITATION

In this section we define a language for statically checked string sanitation (λ_S). The system has regular expression types $\mathsf{stringin}[r]$ whre r is a regular expression. Expressions of this type evaluate to string literals in the language described by r. Operations on expressions of type $\mathsf{stringin}[r]$ preserve this property.

The premier operation for manipulating strings in λ_S is string substitution, which is a familiar operation to any programmer who has used regular expressions. The replacement operation replaces all instances of a pattern in one string with another string; for instance, $\mathtt{lsubst}(a|b, a, c) = c$. In order to compute the type resulting from aubstitution, we also need to compute the result of replacing one language with another inside a given language. Finally, just for convienance, we provide a coerce operation. The introduction of coercion requires handling of runtime errors.

The underlying language λ_P has only one type for strings. We prove that whenever a term is translated from λ_S to λ_P, correctness is preserved. The only exception is in the case of unsafe casts in λ_S, which are unnecessary but are included to demonstrate that the regex library of λ_P may be used to insert dynamic checks whenever even when developers are not careful about using statically checked operations.

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.

$$
\begin{array}{rcll}
\psi & ::= & \psi \to \psi & \text{source types} \\
 & | & \mathsf{stringin}[r] &
\end{array}
$$

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

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

$$
\begin{array}{rcll}
\theta & ::= & \theta \to \theta & \text{target types} \\
 & | & \mathsf{string} & \\
 & | & \mathsf{regex} &
\end{array}
$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{}{[\![\mathsf{rstr}[s]]\!] = \mathsf{str}[s]}\ \textsc{Tr-string}$$

$$\frac{[\![S_1]\!] = P_1 \qquad [\![S_2]\!] = P_2}{[\![\mathsf{rconcat}(S_1, S_2)]\!] = \mathsf{concat}(P_1, P_2)}\ \textsc{Tr-Concat}$$

$$\frac{[\![S_1]\!] = P_1 \qquad [\![S_2]\!] = P_2}{[\![\mathsf{rreplace}[r](S_1, S_2)]\!] = \mathsf{replace}(\mathsf{rx}[r], P_1, P_2)}\ \textsc{Tr-Subst}$$

$$\frac{S : \mathsf{rstr}[r] \qquad \mathcal{L}\{r'\} \subseteq \mathcal{L}\{r\}}{[\![\mathsf{rcoerce}[r'](S)]\!] = \mathsf{str}[s]}\ \textsc{Tr-Coerce-Ok}$$

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

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

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

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

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

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

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

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

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

$$\frac{}{\mathsf{str}[s] \Downarrow \mathsf{str}[s]}\ \textsc{P-E-Str} \qquad \frac{}{\mathsf{rx}[r] \Downarrow \mathsf{rx}[r]}\ \textsc{P-E-Rx}$$

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

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

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

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

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 regular expression r. This distinction between the regular 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 regular expressions r, either $s \in \mathcal{L}\{r\}$ or $s \notin \mathcal{L}\{r\}$. Regular languages are closed under difference, right quotient, reversal, and string homomorphism.*

If any of these properties are unfamiliar, the reader may refer to a standard text on the subject [4].

Definition 2 ($\texttt{lsubst}$)**.** 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 ($\texttt{lreplace}$)**.** 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, reversal and string homomorphism. □

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 definitions 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 $S \Downarrow \mathsf{rstr}[s]$ and $\Psi \vdash \mathsf{rstr}[s] : \mathsf{stringin}[r]$; 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 ensures that well-typed terms of regular string type are in the language associated with their type.

Theorem 9. *Correctness of Input Sanitation for λ_S. If $\Psi \vdash S : \mathsf{stringin}[r]$ and $S \Downarrow \mathsf{rstr}[s]$ then $s \in \mathcal{L}\{r\}$.*

Proof. Follows directly from type safety, canonical forms for λ_S. □

Theorem 10. *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

The main theorem of this paper is Theorem 12. Establishing this result requires an additional theorem (Theorem **??**), which establishes a relationship between canonical forms for the string fragments of λ_S and λ_P.

Theorem 11. *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. □

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 11, $P \Downarrow \mathsf{str}[s]$ implies that $S \Downarrow \mathsf{rstr}[s]$. By theorem 9, this together with the assumption that S is well-typed implies that $s \in \mathcal{L}\{r\}$. □

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

An implementation of a similar system as a type system extension in the programming language Ace is discussed in [2] and [3]. The implementation only supports a special case of replacement where unsafe substrings are simply stripped. In practice, most libraries and frameworks escape command sequences instead of stripping characters; therefore, we plan to extend our implementation with support for the replace operation as described in this paper..

```
#todo: why are these r's here?
#TODO-nrf this is maybe kind-of right bue I think there's a nicer way to do it
@fn
def sanitize(s : string_in[r'.*']):
  return (s.replace(r'"', '&quot;')
           .replace(r'<', '&lt;')
           .replace(r'>', '&gt;'))

@fn
def results_query(s : string_in[r'[^"]*']):
  return 'SELECT * FROM users WHERE name="' + s + '"'

@fn
def results_div(s : string_in[r'[^<>]*']):
  return '<div>Results for ' + s + '</div>'

def main(db):
  input = sanitize(user_input())
  results = db.execute(results_query(input))
  return results_div(input) + format(results)
```

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

5. 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.

6. REFERENCES

[1] A. Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *OSDI'10: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2010.

[2] N. Fulton. Security through extensible type systems. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 107–108, New York, NY, USA, 2012. ACM.

[3] N. Fulton. A typed lambda calculus for input sanitation. Undergraduate thesis in mathematics, Carthage College, 2013.

[4] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[5] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, May 2003.

[6] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular Expression Types for XML. In *ICFP '00*, 2000.

[7] OWASP. Open web application security project top 10.

[8] M. Sulzmann and K. Lu. Xhaskell – adding regular expression types to haskell. In O. Chitil, Z. HorvÃąth, and V. ZsÃşk, editors, *Implementation and Application of Functional Languages*, volume 5083 of *Lecture Notes in Computer Science*, pages 75–92. Springer Berlin Heidelberg, 2008.


```
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.