Statically Typed String Sanitation Inside a Python

Nathan Fulton
Cyrus Omar
Jonathan Aldrich

Carnegie Mellon University

The Problem

Applications use **strings** to build `SQL` commands

```
sql_exec("SELECT * FROM users WHERE" +
    "username = " + input1 + " AND " +
    "password = " + input2)
```

The Problem

Applications use **strings** to build `HTML` commands

```
print("You searched for: " + keyword)
```

The Problem

Applications use **strings** to build `JS` commands

```
print("<script>" +
   "document.getElementById(" +
   "'" + input + "'"  +
   ")" + "..." +
   "</script>")
```

The Problem

Applications use **strings** to build `shell` commands

```
call("cat " + input)
```

Arbitrary strings are dangerous.

Existing Solutions

- Web Frameworks

Existing Solutions

- Web Frameworks
  - may contain bugs

Existing Solutions

- Web Frameworks
  - may contain bugs
- Prepared Statements

Existing Solutions

“Drupal is an open source content management platform powering millions of websites… During a code audit of Drupal extensions for a customer **an SQL Injection was found in the way the Drupal core handles prepared statements**. A malicious user can inject arbitrary SQL queries… This leads to a code execution as well.”

- Stefan Horst, 6 days ago

Existing Solutions

- Web Frameworks
  - may contain bugs
- Prepared Statements
  - may contain bugs

Existing Solutions

- Web Frameworks
    - may contain bugs
- Prepared Statements
    - may contain bugs
- Problem specific parsers

Existing Solutions

“Three of our Sports API servers had malicious code executed on them… This mutation happened to exactly fit a **command injection bug in a monitoring script** our Sports team was using at that moment to **parse and debug their web logs**.”

- Alex Stamos (Yahoo! CISO), two weeks ago

Existing Solutions

- Web Frameworks
  - may contain bugs
- Prepared Statements
  - may contain bugs
- Problem specific parsers
  - may contain bugs

The Goal: A ***general*** approach for specifying and verifying input sanitation procedures, ***with a minimal trusted core***.

Arbitrary strings are dangerous.

Static reasoning about strings is easy!

Regular Expression Types

Python, Java, etc:

string

Lambda RS:

string[regex]

Contributions

- Regular Expression Types corresponding to common string and regex library operations.
- Translation into a language with a bare string type.

Together, these define a **type system extension** which is implemented in the extensible programming language atlang.

Typing Rule for String Literals

If:

- *s* in a string in the language of *r*

Then:

- rstr[s] has type stringin[r].

Typing Rule for String Literals

$$\frac{s \in \mathcal{L}\{r\}}{\Psi \vdash \mathsf{rstr}[s] : \mathsf{stringin}[r]}$$

The Security Theorem

If e has type stringin[r], then e evaluates to a string (denoted rstr[s]) such that s $\in$ L(r).

```
"""this function will remove quotes."""
def sanitize(s : string): s //TODO

def get_user(u : string):
   sql_exec("select * from users where " +
      "username = '" + u + "'")
```

```
"""this function will remove quotes."""
def sanitize(s : string): s //TODO

def get_user(u : string):
   sql_exec("select * from users where " +
      "username = '" + u + "'")

x = "';DELETE FROM users--"
get_user(sanitize(x))
```

```
"""this function will remove quotes."""
def sanitize(s : string): s //TODO

def get_user(u : string[!']):
   sql_exec("select * from users where " +
      "username = '" + u + "'")

x = "';DELETE FROM users--"
get_user(sanitize(x))
^ type error! L(.*) is not in L(!')
```

```
"""this function will remove quotes."""
def sanitize(s : string) -> stringin[!']:
  s.replace(r"'", "")

def get_user(u : string[!']):
   sql_exec("select * from users where " +
      "username = '" + u + "'")

x = "';DELETE FROM users--"
get_user(sanitize(x))
^ OK!
```

Regular Expressions

r ::= a | r·r | r ++ r | r*

Regular Languages

r ::= a | r·r | r ++ r | r*

L(psp) = {psp}

L(ps*p) = {pp, psp, pssp, psssp, ...}

L(a ++ b) = {a, b}

Regexes as Specs

Often Unstated Specifications:

! '

Regexes as Specs

Often Unstated Specifications:

```
    !'
    (a|b|c|...)*
```

Regexes as Implementations

Often Unstated Specifications:

```
    !'
    (a|b|c|...)*
```

Implementations:

```
    replace(!’, "", input)
```

Unstated Assertion:

implementation meets specification.

The Core Language (1 / 2)

Construct	Abstract Syntax	A Python
Concat	`rconcat(e1;e2)`	`e1 + e2`

Substring	`rstrcase(e1;`
	`  e2;`
	`  x,y.e3)`
		`if e1 == "":`
		`  e2`
		`else:`
		`  e3(e1[:1], e1[1:])`

Replace	`rreplace[r](e1; e2)`	`e1.sub(r"r", e2)`

The Core Language (2 / 2)

Concept	**Abstract Syntax**	**A Python**
Coercion	`rcoerce[r](e)`	`e`
Checks	`rcheck[r](e; x.e1; e2)`	`if re.search(r”r”,e) == None:` `e2` `else:` `e1(e)`

λ_{RS}

String Concatenation

rconcat(e; e)

Substrings

rstrcase(e; e; x,y.e)

Substitution

rreplace[r](e; e)

Coercions

rcoerce[r](e)

Checked Casts

rcheck[r](e; x.e; e)

String Concatenation

Recall: if e has type stringin[r] then e evaluates to v and v $\in$ L(r).

String Concatenation

Recall: if e has type stringin[r] then e evaluates to v and $v \in L(r)$.

If:

- e_1 : stringin[r_1]
- e_2 : stringin[r_2]

then:

- concat(e_1; e_2) : stringin[$r_1 r_2$].

String Concatenation

Recall: if e has type stringin[r] then e evaluates to v and v ∈ L(r).

S-T-CONCAT

$$\frac{\Psi \vdash e_1 : \mathsf{stringin}[r_1] \qquad \Psi \vdash e_2 : \mathsf{stringin}[r_2]}{\Psi \vdash \mathsf{rconcat}(e_1; e_2) : \mathsf{stringin}[r_1 \cdot r_2]}$$

Example Typing Derivation

$$\dfrac{\dfrac{a \in \mathcal{L}\{a*\}}{\Psi \vdash \mathsf{rstr}[a] : \mathsf{stringin}[a*]} \qquad \dfrac{b \in \mathcal{L}\{b*\}}{\Psi \vdash \mathsf{rstr}[b] : \mathsf{stringin}[b*]}}{\Psi \vdash \mathsf{rconcat}(r; \mathsf{rstr}[a])\mathsf{rstr}[b] : \mathsf{stringin}[a * b*]}$$

Substrings

```
""" S = state code then D.O.B. """
def get_state(s : stringin[(a-z0-9)*]):
   rstrcase(s; ''; x + rstrcase(y; ''; x))
```

Substrings

```
get_state("WI1956")
```

Substrings

```
get_state("WI1956")
```

$$\Downarrow$$

```
rstrcase("WI1956"; ''; x + rstrcase(y; ''; x))
```

Substrings

```
get_state("WI1956")
⇓
rstrcase("WI1956"; ''; x + rstrcase(y; ''; x))
⇓
"W" + rstrcase("I1956”; ''; x)
```

Substrings

```
get_state("WI1956")
```

$\Downarrow$

```
rstrcase("WI1956"; ''; x + rstrcase(y; ''; x))
```

$\Downarrow$

```
"W" + rstrcase("I1956”; ''; x)
```

$\Downarrow$

```
"W" + "I" = "WI"
```

Substrings

“Get the first n characters of a string s”

Substrings

“Get the **first** character of a string s”

“Get everything after the first character of s”

Substrings

“Get the **first** character of a string s”

```
lhead(r)            = lhead(r, ε)
lhead(ε, r’)        = ε
lhead(a, r’)        = a
lhead(r1·r2, r’)    = lhead(r1, r2)
lhead(r1 + r2, r’) = lhead(r1, r’) + lhead(r2, r’)
lhead(r*, r’)       = lhead(r’, ε) + lhead(r, ε)
```

Substrings

“Get the **first** character of a string s”

```
lhead(r)              = lhead(r, ε)
lhead(ε, r′)          = ε
lhead(a, r′)          = a
lhead(r1·r2, r′)      = lhead(r1, r2)
lhead(r1 + r2, r′) = lhead(r1, r′) + lhead(r2, r′)
lhead(r*, r′)         = lhead(r′, ε) + lhead(r, ε)
```

“Get everything after the first character of s”

$\delta_a(r) + \delta_b(r) + \delta_c(r) + \ldots$

Substrings

Observation: If s $\in$ L((a-z)*(0-9)) then get_state(rstr[s]) $\Downarrow$ rstr[t] such that t $\in$ (a-z0-9)*.

Substrings

Observation: If s ∈ L((a-z)*(0-9)) then get_state(rstr[s]) ⇓ rstr[t] such that t ∈ (a-z0-9)*.

S-T-Case

$$\frac{\Psi \vdash e_1 : \mathsf{stringin}[r] \qquad \Psi \vdash e_2 : \sigma \qquad \Psi, x : \mathsf{stringin}[\mathsf{lhead}(r)], y : \mathsf{stringin}[\mathsf{ltail}(r)] \vdash e_3 : \sigma}{\Psi \vdash \mathsf{rstrcase}(e_1; e_2; x, y.e_3) : \sigma}$$

On the precision of rstrcase

Note that lhead(r)·ltail(r) ≠ r.

On the precision of rstrcase

Note that lhead(r)·ltail(r) ≠ r.

Example: Choose r = (ab)+(cd), so “ad” ∉ L(r).

Note that:

```
lhead(r) = a + c
ltail(r) = δ_a(r) + δ_c(r)
         = b + d
```

Therefore, “ad” ∈ L(lhead(r)·ltail(r)).

String Replacement

$$\textsc{S-E-Replace}\quad \frac{e_1 \Downarrow \mathsf{rstr}[s_1] \qquad e_2 \Downarrow \mathsf{rstr}[s_2] \qquad \mathsf{subst}(r; s_1; s_2) = s}{\mathsf{rreplace}[r](e_1; e_2) \Downarrow \mathsf{rstr}[s]}$$

subst(r; s1; s2) reads “substitute s2 for r in s1”

String Replacement

S-T-Replace

$$\frac{\Psi \vdash e_1 : \mathsf{stringin}[r_1] \qquad \Psi \vdash e_2 : \mathsf{stringin}[r_2] \qquad \mathsf{lreplace}(r; r_1; r_2) = r'}{\Psi \vdash \mathsf{rreplace}[r](e_1; e_2) : \mathsf{stringin}[r']}$$

String Replacement

Key Fact: lreplace and subst correspond:

subst(r, s1, s2) is in lreplace(r, r1, r2)

where:

- s1 ∈ r1, and
- s2 ∈ r2.

String Replacement

subst(r, s1, s2) is in lreplace(r, r1, r2).

This does **not** entail a definition of lreplace given a definition of subst.

Saturation

```
replace("ee", "Kleeene", "e")
```

replace ee in "Kleene" with e

```
  = “Kleene”
```

Translation

$$\textsc{Tr-Concat}\quad \frac{[\![e_1]\!] = \iota_1 \qquad [\![e_2]\!] = \iota_2}{[\![\mathsf{rconcat}(e_1; e_2)]\!] = \mathsf{concat}(\iota_1; \iota_2)}$$

Translation

Translation defines either an embedding (as a language extension) or, alternatively, an erasure.

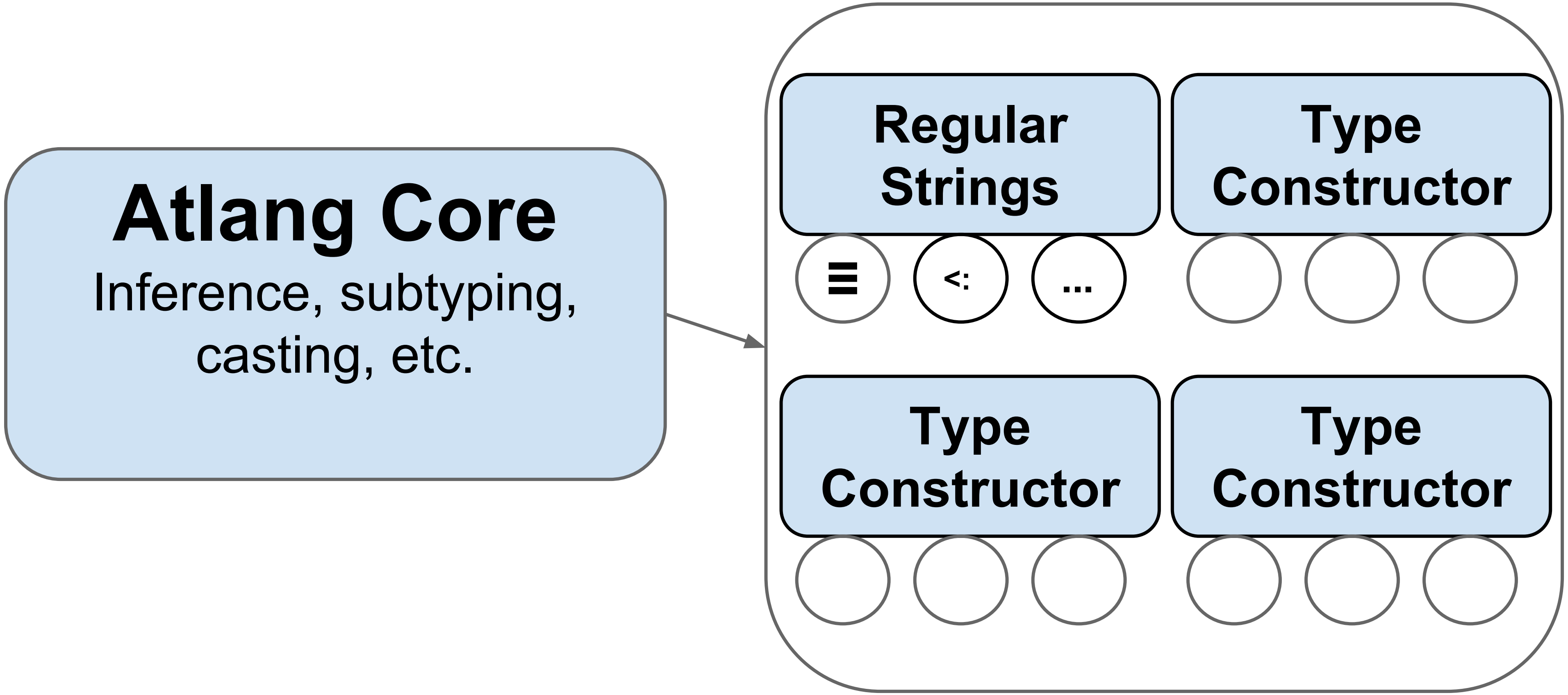
```python
from atlib import fn, stringin

@fn
def sanitize(s : stringin[r'.*']):
  return (s.replace(r'"', '&quot;')
           .replace(r'<', '&lt;')
           .replace(r'>', '&gt;'))

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

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

@fn
def main():
  input = sanitize(user_input())
  results = db_execute(results_query(input))
  return results_div(input) + format(results)
```


Atlang Core
Inference, subtyping, casting, etc.
Regular Strings
≡
<:
...
Type Constructor
Type Constructor
Type Constructor

Conclusions

Constrained String Types are a *general* approach for specifying and verifying input sanitation procedures.

Unlike other approaches, constrained strings only require a minimal trusted core.

Future Work

1. Implement a static analysis and verify a realistic query builder.
2. Application of replacement operation to program repair in dynamic logic over trace semantics.
    - replacement on hybrid regular programs.
3. Explore other privacy & security applications of extensible type systems.

```
  def __init__(self, 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, stringin):
      left_rx = left_t.idx
      if isinstance(right_t, stringin):
        right_rx = right_t.idx
        return stringin[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, stringin):
      raise atlang.TypeError("...", node)
    exp_rx = exp_t.idx
    return stringin[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]))

  # check and strcase omitted

  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
```