Security Through Extensible Type Systems

Nathan Fulton
Carthage College, Carnegie Mellon University
nfulton@carthage.edu

Abstract

Researchers interested in security often wish to introduce new primitives into a language. Extensible languages hold promise in such scenarios, but only if the extension mechanism is sufficiently safe and expressive. This paper describes several modifications to an extensible language motivated by end-to-end security concerns.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifiers - Extensible languages

Keywords Extensibility, Security

1. Introduction

Researchers involved in security research use language primitives to address software verification problems. However, many high-profile vulnerabilities have still not been addressed in popular languages. For example, although improperly sanitized user input is the top cause of vulnerabilities in mobile and web applications [3], popular languages have not incorporated language primitives that guarantee that input has been sanitized. Extensibility mechanisms capable of modularly expressing and verifying such primitives could help decrease the gap between research and practice.

Contributions Ace is an extensible language being developed by our group. This paper describes an extension within Ace capable of statically verifying that a string is in a specified language. Implementing this extension required adding support for subtyping and type-casting to the Ace extension mechanism itself. We use the idea of type-directed compilation [5] to enforce a specified correspondence between the source and target type systems.

These contributions demonstrate the efficacy of using an extensible type system to verify mission critical portions of a program.

SPLASH'12, October 19–26, 2012, Tucson, Arizona, USA.
ACM 978-1-4503-1563-0/12/10.

2. Constrained Strings in Ace

Although existing frameworks and libraries provide sanitation mechanisms, empirical studies show that developers misuse these tools [4]. This misuse suggests the need for a simple, universal language primitive that enforces best practices for string sanitation.

We address this need by providing a type system extension capable of statically checking that that a string is in a regular language. The inputs to insecure functions are constrained using a statically specified constrained string type. Our type system (see Figure 1, described later) provides a statically checked mechanism for converting strings into appropriately typed constrained strings. Values that have the type String must be cast to the specified type before use. Our type system provides a statically checked mechanism for achieving this conversion and updating the language a string belongs to when operations are performed on it.

Ace Ace[1] compiles programs from a typed source language to a typed target language based on specifications associated directly with type definitions, written in a "type-level language". This approach is called "active compilation" [2].

Listing 1 is an Ace program demonstrating the constrained string extension. Ace functions are declared generically, then specialized with a `backend` (specifying the target language) and input types. Functions are compiled and specialized, one function at a time, on lines 4 and 9. Return types are inferred from input types. When `sanitize_query` is compiled, our typechecker ensures that the argument to `run_query` matches the input type specified on line 2.

```
1  CS = ConstrainedString
2  @ace.fn
3  def run_query(input):
4      return execute_query(input)
5  run_query = run_query.compile(backend,
6                                CS("(a-z0-9)+"))
7  @ace.fn
8  def sanitize_query(string):
9      return run_query(
10         `remove("^(a-z)+")`(string))
11 sanitize_query = sanitize_query.compile(
       backend, Str, run_query.type)
```

Listing 1. An Ace program using constrained strings.

[1] https://github.com/cyrus-/ace

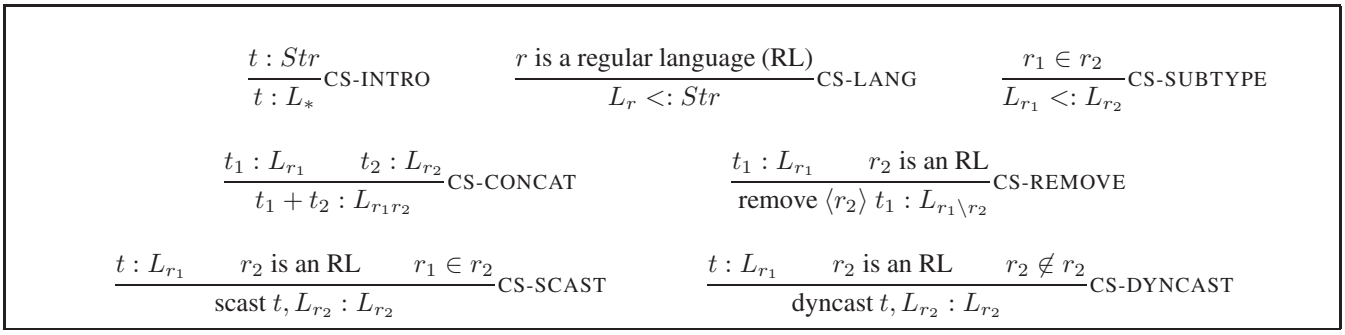

Figure 1. Some of the type checking rules relevant to the `ConstrainedString` extension.

The rest of Listing 1 is an application of the typechecking rules for constrained strings, as described in Figure 1. The expression surrounded by back-ticks on Line 7 uses static evaluation to parameterize the remove function with a regular expression. Calling the resulting function removes all substrings matching the constrained string type with the static string (see `CS-LANG` in Figure 1.) In this case, the value returned by `sanitary_query` has type `CS("(a-z)+")`. Subtyping between constrained strings permits this expression to be used where the larger type, `CS("(a-z0-9)+")`, is specified.

Constrained Strings Constrained strings are a family of types parameterized by the regular languages. We believe the checking rules provided in figure 1 describe a sufficiently expressing relation for addressing these needs:

- The `CS-LANG` rule establishes a correspondence between each regular language r and its constrained string type L_r.
- The `CS-SUBTYPE` defines subtyping as language inclusion: if the strings in one language are a subset of the strings in another language, then the smaller language is a subtype of the larger.
- The `CS-INTRO` rule admits arbitrary strings into the typing relation for constrained strings.
- The `CS-REMOVE` rule removes all substrings of t_1 matching r_2. Most sanitation algorithms have this form. The `CS-CONCAT` rule can be used to convert between constrained string types.
- The type casting rules are straight forward, but the evaluation rule for `CS-DYNCAST` inserts a runtime check.

Subtyping and Casting We modified Ace's extensibility mechanism so that backends and extensions may define subtyping and checked downcasts.

3. Checking Extensions to Ace

To have confidence in the security guarantees offered by an extension, we must check that the extension itself is correct. As an example, if Ace is compiling to C, then the constrained string type must enforce the typing relation given in Figure 1 and also generate code of type `char*`. The strategy for mechanizing this test follows from the design of Ace. Each expression generated by the compiler is tagged with an expected type, specified by the extension writer. The Ace type checker ensures that each generated expression has the expected type.

4. Related Work

Several languages provide mechanisms for extensibility. Ace uses a novel extension mechanism with type-directed specifications called Active Type Checking and Translation.

Regular expression types are used in several languages. Our subtyping extension is motivated by XDuce [1]. We are unaware of a predecessor to `CS-REMOVE`.

5. Future Work

Only portions of the Ace type system used by a program get checked during compilation. We plan to statically check these extensions be using a dependently-typed type-level language.

Acknowledgments

The author is advised by Cyrus Omar and Jonathan Aldrich and supported by a grant from the Army Research Office under Award No. W911NF-09-1-0273.

References

[1] Hosoya and Pierce. XDuce: A Statically Typed XML Processing Language. ACM Transactions on Internet Technology, 3(2):117-148, May 2003.

[2] Omar, Cyrus. Active Type Checking and Translation. SPLASH SRC 2012.

[3] OWASP. Top Ten Project. 2010.

[4] Scholte, Balzarotti, Kirda. Quo vadis? A study of the evolution of input validation vulnerabilities in Web applications. FCDS 2011.

[5] Tarditi, Morrisett, Cheng, Stone, Harper, Lee. Til: a type-directed optimizing compiler for ML. PLDI '96.