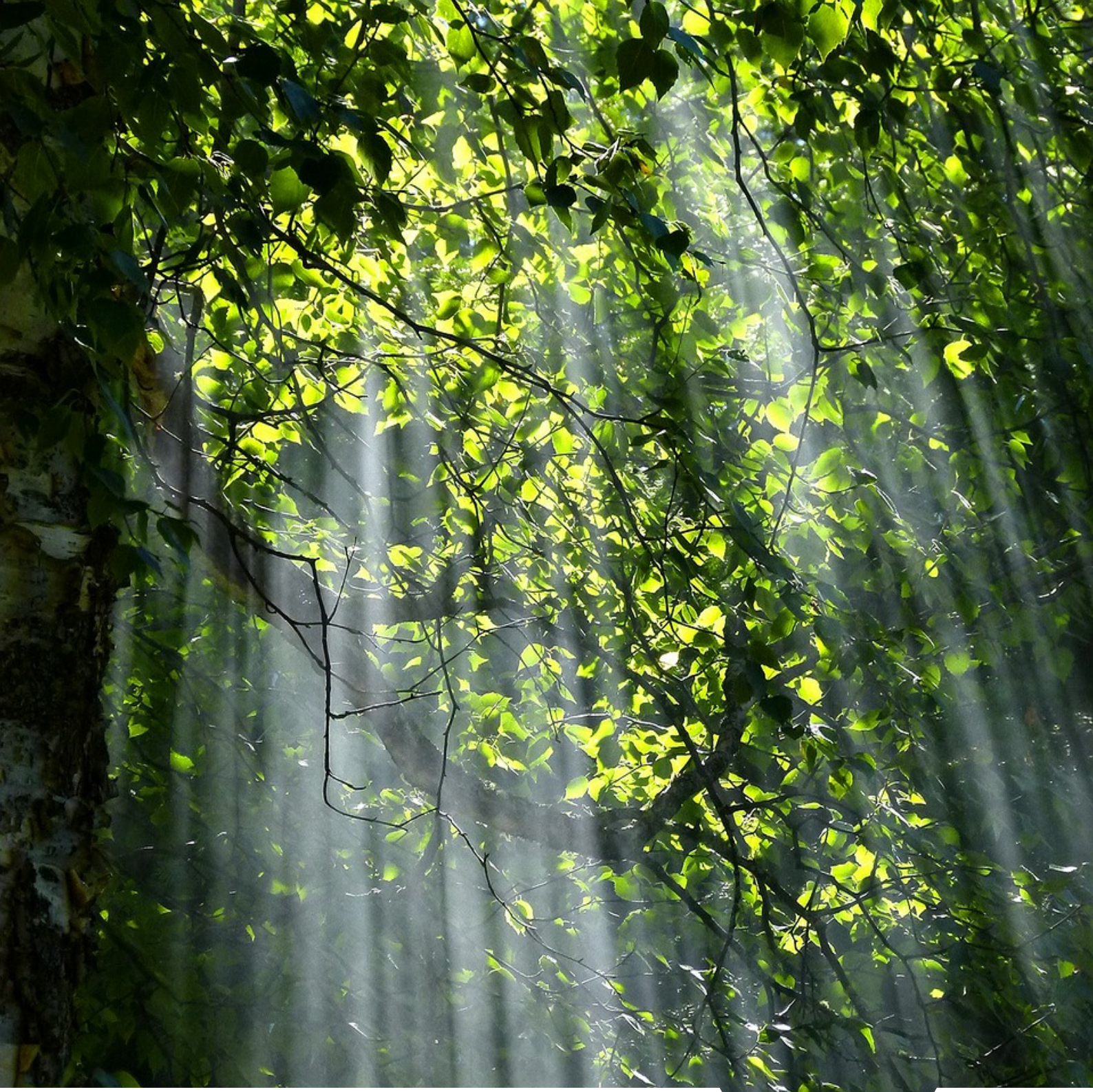

Regular Expression Refinement Types

WARWICK

Project Specification

Adam Williams

Introduction

Within information security, entire classes of application vulnerabilities arise due to problematic user input handling (Christey and Martin, 2007). This covers cross-site scripting (XSS), injection (SQL, LDAP, etc), insecure deserialisation and file inclusion vulnerabilities – all of which are regularly discovered in major software products.

There are many existing products that aim to detect problems within an application. Sadowski et al. (2018) describe the benefits of static analysis tooling deployed within Google which allow checks for common issues to be performed as part of the compilation process. The authors explain one of the main challenges with developer adoption as *trustworthiness* "users do not trust to results due to, say, false positives" and highlight the importance of reporting issues early: "survey participants deemed 74% of the issues flagged at compile time as real problems, compared to 21% of those found in checked-in code".

In the context of application security, tooling can be broadly categorised into the two broad areas of DAST (dynamic application security testing) and SAST (static application security testing) tooling. SAST tooling can analyse a codebase at rest and lends itself to generating much more immediate results that can take the entire codebase into account. DAST tooling allows for assessment from a black-box perspective but is much more limited. Research into the effectiveness of DAST-based scanning tools by Doupé et al. (2010) drew the conclusion that commonly available tools of this kind often failed to crawl more complex parts of an application which resulted in decreased coverage from a security perspective.

Existing static analysis tools can provide useful warnings that help prevent introduction of vulnerabilities. In the .NET ecosystem, tools such as *Roslyn Security Guard* can detect e.g. injection vulnerabilities by tainting user input and then using information flow analysis – first described in Denning and Denning (1977) – to determine when unsafe input is passed to a dangerous *sink* function (Arteau, 2016). Of course, this approach is limited. All user input is deemed unsafe by virtue of being user input and prior validation (as is commonly performed using regular expressions) is not taken into account when deciding if an alert needs to be raised or not. This leads to false positive reports where a program is secure by virtue of already performing validation to prevent a vulnerability.

The main objective of this project is to explore the use of a type system which uses regular-expression based refinement types applied to user input. A refinement type in the context of a type system is a type that is subject to a particular predicate (Pierce, 2002, p. 207). By considering flow of data that belongs to a refinement type for a particular pattern, it will be possible to make inferences about vulnerabilities that may be present in a codebase based on declared safe argument types. Use of refinement types in this way provides for a more informed evaluation of a particular risk than base types alone and should therefore enable reporting of fewer false positive issues.

```
function LookupName(userId: /[A-Za-z0-9]+/): string {
    var name: string = GetNameByUserId(userId); // safe
    GetNameByUserId("'; DROP TABLE users;"); // type error
    return name;
}

function GetNameByUserId(query: /[^`"']+/): /[A-Za-z ]+/ { /* database lookup.. */ }
```

Listing 1: Example code illustrating a potential syntax. `userId` and `query` use the refinement type.

Objectives

Primary Objectives

- Formalise a type system that supports types predicated with a regular expression pattern that elements of the refined type will satisfy (be matched by).
    - Explore the consequences of typical string operations (e.g. concatenation) and define the type of their return value when applied to elements of the regular expression type.

- At minimum, this should allow for simple functions to be declared that can safely accept/return a particular regular expression input[1].
- Evaluate the rate of false positives when compared to existing static analysis

- Implement such a type system that can guarantee type safety, built against a simplified proof-of-concept language.
  - Test the implementation against a variety of test cases. The testing strategy should make use of automated unit tests, and manual system testing considering both general expected input as well as any relevant "edge-cases" that need to be handled.

Additional Objectives

- Apply the theory explored in the primary phase of the project to produce a type analysis tool which works against type annotations applied to a commonly-used language such as C# or Scala. This tooling could be integrated into an IDE or CI pipeline.

ℹ Primary objectives are expected to be completed during the lifetime of the project. Additional objectives are identified as potential goals to pursue beyond the original scope of the project, if time permits.

Schedule

Time Window	Work
October 1st – October 14th	Specification completion, research into prior related works. Study of elementary programming language and type system theory (e.g. simply typed λ-calculus, SLam).
October 15th – October 28th	Begin writing background for report, work on formalisation of regular expression refinement type. **Deadline**: CS353 presentation, 24th October
October 29th – November 11th	Explore and document properties of type system. Begin implementation of ideas to produce a concrete proof-of-concept.
November 12th – November 25th	Completion of progress report, continued implementation work.
November 26th – December 9th	Testing of implemented proof-of-concept. **Deadline**: CS915 coursework, 26th November
December 10th – January 6th	Slack time (to use if behind schedule, else to make a start on year scheduled in 2019).
January 7th – January 20th	Finalise testing of implementation, write-up test cases. **Deadline**: CS324 coursework
January 21st – February 3rd	Evaluate false positive rates against existing systems based exclusively on taint tracking.
February 4th – February 17th	Report work, project presentation preparation
February 18th – March 3rd	Project presentation preparation, report work
March 4th – March 17th	Project presentation delivery, report finalisation.

Table 1: Projected work by time period. Deadlines for other modules included where known.

[1] As a simplified example, an `unsafe_shell_exec` function might safely be able to accept any input that matches `^[^`]$`

Table 1 provides a breakdown of the project time into periods for each fortnight, along with the expected work to be completed. A meeting will be scheduled for each week to discuss progress and any road-blocks that arise with the project supervisor

Methodology

Software Engineering

This project includes an element of software engineering. Namely, the design and implementation of a proof of concept language which supports a type system incorporating regular expression refinement types.

This implementation work will be carried out in an Agile fashion to fit in with the short timescales inherent to the project and allow for greater flexibility. Time periods in the schedule which involve development work will be treated as a number of week-long development sprints with priorities formalised prior to the commencement of each period. Progress will be reviewed in weekly supervision meetings.

Testing will be automated via the use of unit testing to ensure that specific components function according to their specification in isolation. Where appropriate, integration and system testing can be used to test the solution as a whole (for example, an entire program as a test case would fit into this part of the testing process).

Evaluation

Towards the end of the implementation phase, the false positive rate of the proof of concept tool should be compared with that of existing static analysis tooling based on taint tracking alone.

Logically equivalent test cases should be built for each tool under evaluation in the necessary programming language. Both safe and unsafe function invocations should be included in each test case for completeness.

Resources and risks

The project is reliant on a number of resources. Use of these resources is subject to the risks outlined in table 2. These risks should be evaluated and managed to minimise any potential impact on the project.

Resource	Applicable risk(s)	Impact
VCS hosting: GitHub Storing and tracking code and report changes	Loss of availability due to outage	Minimal, *git* is decentralised so copy of files always available locally and at off-site backup
Report authoring: LaTeX Writing and compiling the report, tracking bibliography	Obsolescence	Unlikely, TeX tooling has been used for decades. Even if particular packages ceased working, the bulk of the content would still be accessible as plain text.
C# analysis: Roslyn libary Fulfilling the additional objective by analysing C# code	Loss of availability due to license change	Minimal. Even if Roslyn's OSS status changes, there is no requirement to integrate with C#, other languages would illustrate the potential just as well. This would also not impact a primary project objective.
Self Project work	Illness, coursework deadlines	Minimised by scheduled slack time and identification of applicable coursework deadlines.

Table 2: Resources and associated risks.

Legal, social, ethical and professional issues

As a project with some security motivation, it is possible that legal, social, ethical and professional issues will arise. In particular, evaluation of existing static analysis tooling must be performed with care to ensure that use of any particular external test cases is permitted by the *Copyright, Designs and Patents Act 1988* within the UK.

Additionally, in order to comply with the *Computer Misuse Act 1988*, any static analysis of external code should only be conducted with permission. Discovered issues should be disclosed responsibly.

References

Arteau, P. (2016), 'Roslyn Security Guard'.

Christey, S. and Martin, R. A. (2007), 'Vulnerability type distributions in CVE'.

Denning, D. E. and Denning, P. J. (1977), 'Certification of programs for secure information flow', *Communications of the ACM* **20**(7), 504–513.

Doupé, A., Cova, M. and Vigna, G. (2010), Why Johnny can't pentest: An analysis of black-box web vulnerability scanners, *in* 'International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment', Springer, pp. 111–131.

Pierce, B. (2002), *Types and Programming Languages*, The MIT Press.

Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L. and Jaspan, C. (2018), 'Lessons from building static analysis tools at google', *Commun. ACM* **61**(4), 58–66.
URL: *http://doi.acm.org/10.1145/3188720*