Regular Expression Refinement Types

Project Report

Adam Williams (3rd year Computer Science)

Michael Gale (supervisor)

WARWICK

Table of Contents

Acknowledgements

I would like to thank my supervisor, Michael Gale, for his advice throughout this project and for shaping its direction from the beginning. His feedback has always been comprehensive and insightful, and his dedication to both teaching and supervision is clear to me.

Next, I wish to thank my former colleagues at MWR InfoSecurity. In particular: Kostas Lintovois, my mentor during my time at MWR, taught me a great deal about practical security assurance work and cemented my knowledge of some of the motivating vulnerabilities we discuss in this report. Donato Capitella and James Coote were a pleasure to work with throughout the course of our project creating a platform for security-focussed training "labs"–a project which greatly improved my knowledge of software engineering. And finally, thanks to the various consultants I worked with in the office and on-site: Mohit, James, Dennis, Alex, Amar, Connor and many others.

Within Warwick, I am grateful to Mat Mannion and the teams within IT Services for their support of my security work and exposing me to a range of new programming languages and technologies.

Whilst often overlooked, formatting is an essential part of any report, and I must gratefully acknowledge I have received from the TeX.SX community and LaTeX package maintainers who have answered my numerous questions.

Lastly, sincere thanks are due to my friends and family for their support during the year.

Keywords

Abstract

Entire classes of modern web application vulnerabilities arise due to problematic user input handling. This includes cross-site scripting (XSS), *injection* issues (SQL, LDAP, etc), insecure deserialisation and file inclusion vulnerabilities – all of which are encountered by information security firms on a regular basis in application assessments. This project explores the use of regular expressions as refinement types for constrained data in order to model user input validation. We formalise the type system of such a language and implement it. We then compare our system to and evaluate it against other, existing approaches by considering false positive and negative rates with a series of test cases.

Motivation   CHAPTER 1

Over the last decade, use of web and thick client applications globally has greatly increased. People increasingly access services online–to manage their finances (Jayawardhena and Foley, 2000), use government services (Fox, 2010) and communicate using social media (Boulianne, 2015). On a daily basis users entrust these systems with maintaining the privacy of their personal information and safeguarding their finances. With the transition from a web centred primarily around exchanging documents to a platform for deploying complex applications, security is increasingly important.

1.1. Application Security

Application security is of particular interest because it transcends the underlying infrastructure on which applications are deployed. Despite the popularity of cloud IaaS (infrastructure-as-a-service) and PaaS (platform-as-a-service) offerings which can substantially improve infrastructure security, applications will continue to be vulnerable.

The predominant cause of these vulnerabilities is improper user input handling (Schneier, 2011). The *Open Web Application Security Project* (OWASP) regularly publishes a list of the top ten most critical security issues, and a subset of these are described below (OWASP, 2017):

Injection Covers query injection, where user input is improperly interpolated into a e.g. database Structured Query Language (SQL) query or a directory LDAP search. Malicious user input can retrieve or modify sensitive data, bypass authorisation controls and (in some cases) run arbitrary code (Stuttard and Pinto, 2011, p. 291).

Broken Access Control Covers vulnerabilities such as insecure direct object access (IDOR) where authorisation checks are not implemented consistently and the user's request is trusted in one or more parts of the application (Stuttard and Pinto, 2011, p. 257). Also includes path traversal vulnerabilities where user input is used to construct a filesystem path and a malicious actor can include e.g. `../` to traverse up one level.

XML External Entities (XXE) A vulnerability which involves improperly handling user encoded in the XML interchange format. XML supports functionality to reference entities from an external location. Exploitation may allow arbitrary files to be read, sensitive data to be exposed or code to be executed (Stuttard and Pinto, 2011, p. 384).

Cross-Site Scripting (XSS) When user input is not properly handled in the context of a HTML web page or inside JavaScript, a user can provide malicious input that can run arbitrary code on the client. This can be used by an attacker to steal or misuse a victim's session (Stuttard and Pinto, 2011, p. 431).

Insecure Deserialization Results from insecurely converting user input into a language object.

Many of these vulnerabilities–and the necessary techniques for preventing them–have been well-established for a number of years, and yet they continue to be regularly discovered in production applications (Schneier, 2011, p. 2).

1.1.1. Examples

From the above classes of vulnerabilities, we will focus on a subset of motivating examples. In the context of a Java Enterprise Edition (EE) application these will include SQL injection, LDAP injection and path traversal.

SQL Injection

Structured Query Language (SQL) is used with relational database systems to create, retrieve, update and delete data. SQL injection falls into the *injection* category of vulnerabilities above. Generally, it arises when SQL queries concatenate or interpolate user input into a query command. An attacker is able to craft user input that "breaks out" of e.g. a string literal and execute arbitrary SQL. This could lead to exfiltration of sensitive data, destruction of data or violation of data integrity. SQL injection is typically mitigated using *prepared statements* (Stuttard and Pinto, 2011), which allow placeholders to be set-up in the query and then bound to meaningful values at execution time. By using this technique to bind user input to specific placeholders, the possibility to inject SQL is removed.

The example below shows a Java function, `lookupUserNameByToken`, which performs a query to retrieve data from a USERS table based on a user-provided `token` argument. Instead of using a prepared statement, once the query is created, the token input is concatenated directly into the where clause.

```java
public String lookupUserNameByToken(String token) throws SQLException {
    Statement stmt = connection.createStatement();
    String query = "select * from users where ";
    ⚠ query += " token = '" + token + "'" ;
    ResultSet rs = stmt.executeQuery(sql);
    if (rs.next()) {
        return rs.getString("name");
    }
    return null;
}
```

An attacker able to directly specify a token, could provide an input string containing a single quote `'` character to break out of the string literal in the query.

For instance, providing a token `' or 1=1 --` would create the following query:

```sql
select * from Users where token = '' or 1=1 -- '
```
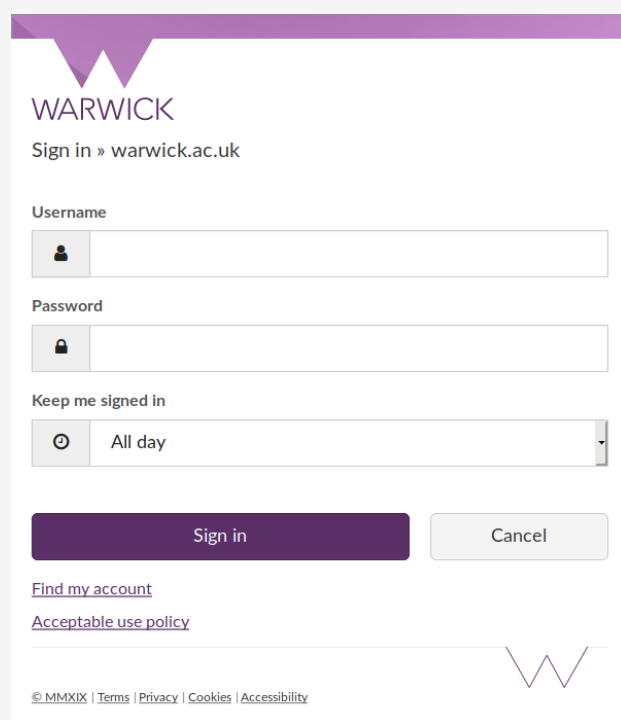
> ⓘ   The `--` token is used to create a comment until the end of the query. This ensures no syntax error is introduced when the Java code ends the string after appending the user input.

This query would return all users in the database.

LDAP Injection

The Lightweight Directory Access Protocol (LDAP) is a networked protocol designed for querying directory services. These services are used for storing and organising network entities, such as users and devices. Microsoft's Active Directory (AD) is a commonly deployed enterprise solution for storing user information and handling network authentication.

🖼 **Figure 1.1:** LDAP is used internally at Warwick to power the single sign-on system

LDAP queries are written in a compact manner using symbols for each operation and parentheses for each clause:

```
(&(warwickUniId=1510654)(sn=Williams))
```
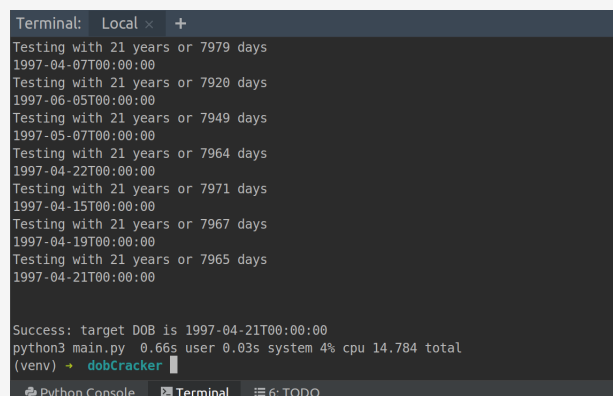
This query retrieves the user with attribute `warwickUniId` set to value 1510654 *and* surname (sn) set to value "Williams". The `&` operator specifies conjunction to be applied to the nested filters.

Much like SQL injection, LDAP injection arises when user-input is concatenated into a filter name or value:

```
public string GetSurnameForUser(string username)
{
    var directoryEntry = new DirectoryEntry("LDAP://acme.corp");
    var searcher = new DirectorySearcher(directoryEntry);
    ⚠ searcher.Filter = "(cn=" + username + ")";
    var result = searcher.FindOne();
    return result.Properties["sn"][0].ToString();
}
```

The impact is generally more limited, but due to the support for wildcard characters it is possible to leak other attribute values by testing for a character at a time. Numeric values or dates can be exfiltrated using a binary search approach.

🖼 Figure 1.2: Injection into a conjunctive query can be used to exfiltrate data

Note that despite network latency, we can still leak a value in a short period of time (< 15s) by repeatedly querying an oracle with less than/greater than filters and observing the response.

The best practice for mitigating this vulnerability involves using an encoder that is able to escape LDAP control characters (for example, right parenthesis is replaced with \29). This ensures that user input is then not considered part of a query.

Path Traversal

Many applications read and write files to the filesystem. Problems arise when user input is permitted to influence the destination path.

Consider a Java function which is designed to persist user preference information in a text file:

```
public static void writeUserPreference(String username, Preference pref) {
    try {
        ⚠ String path = "/tmp/prefs/" + username;
        Files.writeString(Paths.get(path), pref.toString());
    } catch(IOException e) {
        e.printStackTrace();
    }
}
```

If a user is able to influence the username passed as an argument to the function and choose a malicious name, such as the string `../../etc/ssh/sshd_config`, they will be able to write to files on the filesystem outside of the location which the developer intended. This issue can arise with both write and read operations. Depending on which files are available to the application, this vulnerability can lead to a full system compromise.

Path traversal vulnerabilities have been reported in software for at least 20 years but still regularly make appearances in modern applications. As an example, CVE-2019-1002101 was disclosed in March 2019. This vulnerability impacts Kubernetes, a popular container orchestration system developed by Google. The root cause is a directory traversal vulnerability arising due to the system trusting unsafe input from an archive file.

1.2. Static Analysis

Static analysis tools can detect some classes of vulnerability by examining source code. Within application security, static analysis tools are grouped under the *Static Application Security Testing* SAST denomination in contrast to *Dynamic Application Security Testing* (DAST) tools which operate at runtime and observe the behaviour of a running system. SAST tools work by either independently parsing the developer's code or analysing an *abstract syntax tree* (AST) produced by the language toolchain[1] and using a series of inspections to check and log common problems. Some more advanced tools use *taint tracking* combined with information flow analysis to mark and track variables and parameters that have been influenced by user input (Denning and Denning, 1977). Functions or procedures for which it is dangerous to receive user input are marked as *taint sinks*, whereas sources of user input are known as *taint sources*. A list of sinks that are deemed potentially dangerous (such as database query functions) is maintained, and user input flowing to any of these sinks results in an issue being logged.

For the example discussed in section 1.1.1, the `token` argument would be marked as a *taint source*; this indicates that it is either completely or partially influenced by user input. The `executeQuery` method on the `java.sql.Statement` class instance would, in contrast, be recognised as a sink.

As discussed in Sadowski et al. (2018), it is preferable to detect issues in a static fashion–ideally integrated into the build process–to ensure that problems are actioned by developers. The false positive rate should also be minimised to avoid the risk of *alert fatigue*,a problem which occurs

---

[1]In the context of C#, Microsoft have made the *Roslyn* compiler source code available and many static analysis tools are built on top of this open platform.

in a variety of different contexts where the value of alerts is decreased due to a perception that often, there is no real substance to the warnings (Kesselheim et al., 2011).

If reliable and properly implemented, static analysis tooling has been shown to be able to prevent whole classes of bugs from making it into production (Sadowski et al., 2018).

1.3. A Better Approach

Existing static analysis are often unable to evaluate the effectiveness of input validation code that may already be in place–leading to false positives. This is because the tools consider code in isolation and cannot reason about the form of user input. Taint tracking is similarly "binary", where data originates from user input and is assumed dangerous or originates elsewhere and assumed benign.

We describe a system which uses *refinement types* in order to determine whether existing regular expression based input validation is effective. This happens at compile time, using an SMT solver to find situations where input could fail to be matched by a regular expression. This allows for potential security issues to be surfaced during type-checking.

Our work meets the follow formal objectives (first described in appendix A):

1. Formalise a type system that supports types predicated with a regular expression pattern that elements of the refined type will satisfy (be matched by).

   (a) 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.

   (b) At minimum, this should allow for simple functions to be declared that can safely accept/return a particular regular expression input.

   (c) Evaluate the rate of false positives when compared to existing static analysis

2. Implement such a type system that can guarantee type safety, built against a simplified proof-of-concept language.

   (a) 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.

We know of no existing program verification tools capable of verifying regular expression membership properties within a programming language using pre/post-conditions or type annotations–our system is novel in this respect. Section 2.5 discusses some of the existing static analysis tooling and languages that have been developed in more depth.

Background

This chapter discusses the theory underpinning the techniques used in the implementation.

2.1. Regular Expressions

Most programming languages include support for using *regular expressions* to match strings. Formally, regular expressions are a means to specify a *regular* language – equivalent in power to the *deterministic finite automaton* (DFA) and *non-deterministic finite automaton* (NFA).

Some (or all) of the following operations are available to use when building a recognising a language using regular expressions:

R^* (**Kleene-star**) Accept zero or more of the expression R.

R^+ (**Kleene-plus**) Accept *one* or more of the expression R. Equivalent to R^*.

$A|B$ (**Alternation**) Permit expression A *or B*.

AB (**Concatenation**) Accept A followed by B.

R^C (**Complement**) Accept the inverse/complement of the expression R.

We now consider an example regular language. In the UK, the first part of most postcodes matches the format of two letters followed by up to two numbers. For example, `CV8`, `CV4` or `SW1`. If we define the alphabet of uppercase letters $\Sigma_{A-Z} = \{A, B, C, \ldots, Z\}$ and digits $\Sigma_{\mathcal{N}} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ then we can formally describe a language $L(\Sigma_{A-Z}\Sigma_{A-Z}(\Sigma_{\mathcal{N}}|\Sigma_{\mathcal{N}}\Sigma_{\mathcal{N}}))$.

As discussed, the syntax used in most programming languages differs somewhat and offers some convenience features for defining ranges of characters and specifying the desired number of occurrences of a particular expression:

{ } Code Listing Microsoft's .NET includes a comprehensive regular expression library. C#

```csharp
using System.Text.RegularExpressions;
// Match: exactly two occurrences of a character in range A-Z
//        then at least one digit in 0-9 and optionally one more
string pattern = r"[A-Z]{2}[0-9][0-9]?";
var matches = Regex.Matches("CV8", pattern);
```

Regular expressions are used extensively as an initial step when validating user input. For example, the popular web development framework *ASP.NET MVC* natively allows developers to specify validation rules by way of a regular expression attribute. When user data is submitted in e.g. a form, the framework is able to automatically perform validation and reject input which does not match the expression.

{ } Code Listing Entity field validation using regular expressions   C# / ASP.NET MVC

```csharp
public class UserViewModel
{
    // US-style 000-000-0000 phone number
    [RegularExpression(@"^\d{3}-\d{3}-\d{4}$")]
    public string PhoneNumber { get; set; } // property with getter and setter

    // syntactic sugar for a pre-defined email regular expression
    [Email]
    public string Email { get; set; }
}
```

2.1.1. DFAs and NFAs

At their simplest, these automata are state machines which operate on a string by starting in an initial state and processing each character in turn. Depending on the character encountered, the automaton may *transition* to a different state which can be marked as either an accepting or rejecting state. Once all characters are processed, the input string is said to belong to the language if the final state is accepting.

As described in Sipser (2012, p. 35), we can formalise the definition of a DFA in terms of a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where the elements of the tuple are as follows:

Q The set of states in the automaton.

Σ A set of characters known as the *alphabet.*

δ Table of *transitions* between states. Formally, it can be described as a function $\delta : Q \times \Sigma \to Q$ – given a state from Q which the automaton is in, receiving the character in Σ will result in a new state from Q.

q_0 The initial state which the automaton starts in.

F The set of accepting states.

⊠ Figure 2.1: A DFA, accepting the language `(aa)*`
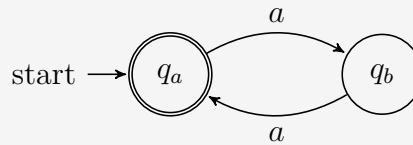
A simple DFA that accepts $2n$ occurrences of the character `a` is shown in figure 2.1. We can define $M = (\{q_a, q_b\}, \{a\}, \delta, q_a, \{q_a\})$ and informally describe the transition function δ as mapping $(q_a,{}'a') \to q_b$ and $(q_a,{}'a') \to q_a$. The automaton alternates between states q_a and q_b every time a new character is processed – whist the length is even, q_a will be active and the string will be considered in the language.

Non-deterministic finite automata are equivalent in language recognition ability to the DFAs discussed above, but use a different processing model (Sipser, 2012, p. 46). There is no requirement to account for every possible character from the alphabet Σ and there is a new concept of an ϵ-transition which can always be followed; such a transition is depicted by an edge labelled ϵ in automata diagrams. NFAs can be thought of as being able to process multiple "paths" in parallel. Figure 2.2 illustrates an NFA using ϵ-transitions to capture *alternation* between the languages g^+ and f^+ as defined at the beginning of section 2.1.

We can formalise the definition of an NFA in terms of a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where the elements of the tuple are described below. Note that the transition function has changed.

Q The set of states in the automaton.

Σ A set of characters known as the *alphabet*.

δ Table defining *transitions* between states. Formally, it can be described as a function $\delta : Q \times \Sigma \cup \{\epsilon\} \to \mathcal{P}(Q)$ – given a state from Q which the automaton is in, receiving the character in Σ will result in a set of possible new states, derived from the powerset (set of possible subsets) $\mathcal{P}Q$.

q_0 The initial state which the automaton starts in.

F The set of accepting states.

Not all languages are regular. For example, the matching parentheses language which accepts strings such as `(())` but not `(, ))` or `((()` can be proven non-regular by contradiction (using the pumping lemma described in Rabin and Scott (1959)). As a result, no DFA, NFA or regular expression[1] can encode the rules of this language.

---

[1] The language *can* however be recognised using a pushdown automata because it belongs to the set of deterministic context-free languages. These are discussed later on.

⊡ Figure 2.2: An NFA, representing the language g+|f+
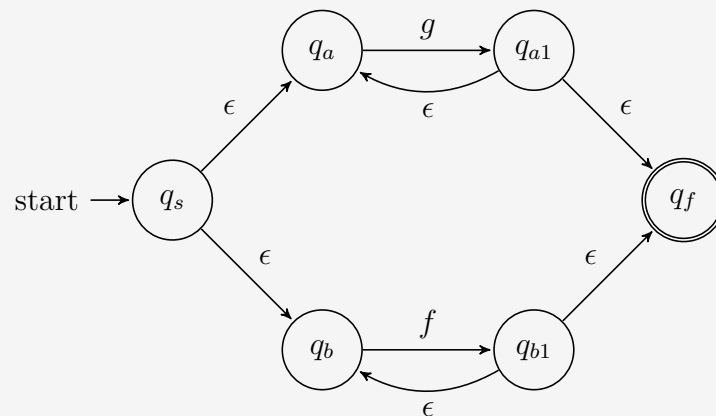
The initial state is q_0. As this automaton is non-deterministic, it can be viewed as "branching" and entering the two states q_a and q_b due to the ϵ-transitions. q_a is used for the **g+** component of the alternation in the original regular expression and, similarly, q_b is used for the **f+** component. Both of these states require at least one occurrence of their respective character before entering the q_{x1} state ($x \in \{a, b\}$) – the only way to proceed to the final accepting state q_f.

2.1.2. Programming Language Support

Most programming languages include a regular expression engine and offer syntax inspired by Perl's regular expressions. These "regular" expressions are often more powerful than the formal regular expressions discussed at the start of section 2.1 and can match non-regular languages.

For example, recall the balanced parentheses language. In PCRE (a library providing a regular expression engine inspired by Perl), the **(?n)** pattern can be used to recursively match using the nth capture group. It is possible to then write an expression which will match balanced parentheses. Micosoft's .NET platform includes an extension to support *balancing group definitions* which would also allow for balanced parentheses to be matched. Regular expressions using these language features and extensions are not *regular* in the formal sense. Throughout this project, we only consider regular expressions according to their formal definition.

2.2. Context-Free Grammars and Parsing

We have briefly discussed regular languages and how they can be represented using regular expressions. In addition, we have introduced DFAs and NFAs which are capable of recognising regular languages.

As illustrated in the Chomsky hierarchy pictured in figure 2.3, languages can fall into a variety of classes. Our language cannot be regular–it needs to be able to parse regular expressions, allowing for nested groups which must balance. As a consequence, we cannot parse the language

in its entirety using a DFA/NFA.

⛰ Figure 2.3: Chomsky hierarchy of grammars
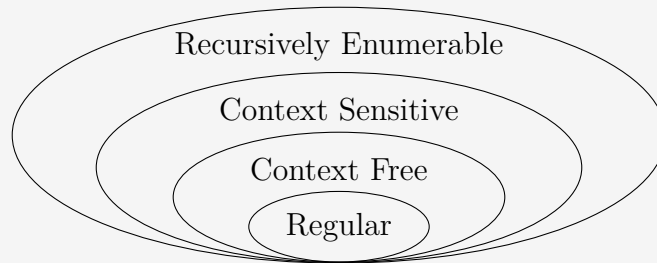
However, the language comprising balanced parentheses lies within the context-free class. We can formally define a context-free grammar to comprise a 4-tuple $G = (V, \Sigma, R, S)$ where the tuple elements are introduced below (Sipser, 2012, p. 102):

V Finite set of *non-terminals*, each of these describes a "sub-language" of G.

Σ Finite set of *terminals* representing string literals (i.e. actual characters in the language), where $V \cap \Sigma = \emptyset$.

R Rules/productions in the grammar.

S The start variable, $S \in V$.

As an example, we define the grammar $G_1 = (\{S\}, \{``('', ``)''\}, R, S)$ where R comprises the following rule:

$S \rightarrow SS \mid (S) \mid \epsilon$

This grammar describes the balanced parentheses language. The rule can be applied as many times as necessary to support an arbitrary number of balanced parentheses. The tree below shows how one can apply the rule to produce the string `()(())`.

$$(S) \text{——} ()$$
$$SS$$
$$(S) \text{——} ((S)) \text{——} (())$$
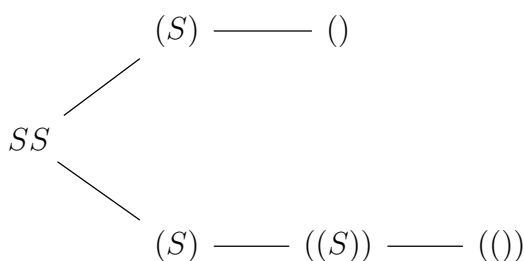
Alternatively, this derivation can be represented in terms of the production rules applied:

$$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()(S) \Rightarrow ()((S)) \Rightarrow ()(())$$

In order to parse context-free languages, we can use an LL-parser. Parsers of this type process the input string from left to right and use the *leftmost derivation*.

2.3. SAT and SMT Solvers

2.3.1. Introducing SAT

The SAT decision problem involves a propositional Boolean logic formula built from a number of variables and operations conjunction $\wedge$, disjunction $\vee$ and negation $\neg$. The question concerns whether the formula is *satisfiable*–i.e. is there a set of *valuations* for each of the variables which results in the overall formula evaluating to true?

If a formula *cannot* be satisfied, there is no valuation for which the formula will evaluate to true. For example, the clauses may contradict each other as in $(a) \wedge (\neg a)$. The example below shows a satisfiable formula comprising 2 clauses and 3 variables:

Can the $(a \vee b \vee c) \wedge (\neg b \vee c)$ propositional logic formula be satisfied?

Yes, set $a = T$, $b = F$, $c = T$. Then clause 1 evaluates to $(T \vee F \vee T) = T$ and clause 2 evaluates to $(\neg F \vee T) = T$. The whole formula evaluates to $T \wedge T = T$.

By exhaustive evaluation, we can prove a problem cannot be satisfied:

Can the $(a \vee b) \wedge (\neg a \vee b) \wedge (\neg b)$ propositional logic formula be satisfied?

No, the clauses are contradictory.
The table below shows exhaustively that no valuation of the variables result in the formula evaluating to true:

a	b	Result
F	F	F
F	T	F
T	F	F
T	T	F

Table 2.1: Enumerating all possible valuations for the variables $\{a, b\}$ used in the formula.

Normal Forms

Examples of formulae up to this point have been given in *Conjunctive Normal Form* (CNF). That is, they are provided as a conjunction of disjunctive clauses. It is possible to convert any Boolean logic formula into this form. To *evaluate* a given formula using a valuation of Boolean variables and values, we simply consider each clause in turn and enumerate the literals within each clause. Once any one literal has been found to be true, the disjunctive nature of the clauses allows us to ignore any remaining terms and move onto the next. Similarly we can skip

all remaining clauses if one clause in a CNF formula is found to evaluate to F, since all clauses must be true.

To solve SAT, we could devise a naïve algorithm considering all possible assignments and evaluating the formula at each stage (as we did in the second example). Clearly such an approach would be inefficient–for Boolean variables with two possible assignment values, this algorithm would work in $\mathcal{O}(2^n)$.

As discussed in Miltersen et al. (2005), it is also possible to convert formulae to *Disjunctive Normal Form* (DNF) using the laws of Boolean algebra. Using the formula from the original example, we can convert to DNF:

$$(a \vee b \vee c) \wedge (\neg b \vee c)$$
$$\equiv ((a \vee b \vee c) \wedge \neg b) \vee ((a \vee b) \vee c) \wedge c \quad \text{Distributivity of } \wedge \text{ over } \vee$$
$$\equiv ((a \vee c) \wedge \neg b) \vee c \quad \text{Complementation of } b \wedge \neg b$$
$$\equiv (a \wedge \neg b) \vee (c \wedge \neg b) \vee c \quad \text{Absorption} : x \wedge (x \vee y) \equiv x$$
$$\equiv ((a \wedge \neg b)) \vee c \quad \text{Distributivity of } \vee \text{ over } \wedge$$

Alternatively, we can use a Karnaugh map to enumerate the possible values for the variables a, b, c and identify groupings to string together in a disjunction:

bc a	00	01	11	10
0	0	1	1	0
1	1	1	1	0

Here, the red group corresponds to the c clause, and the blue group to the $(a \wedge \neg b)$ clause.

A more interesting example might comprise a formula of structure $(a \vee b) \wedge (c \vee d) \wedge (e \vee f)$, which is converted to the lengthy DNF formula below:

$$(a \wedge c \wedge e) \vee (a \wedge c \wedge f) \vee (a \wedge d \wedge e) \vee (a \wedge d \wedge f) \vee (b \wedge c \wedge e) \vee (b \wedge c \wedge f) \vee (b \wedge d \wedge e) \vee (b \wedge d \wedge f)$$

This is of some note, because SAT restricted to formulae in DNF can be solved in linear time using the procedure below:

- For each clause in the formula, check each literal and keep a note of whether it is negated or not

  - If a clause contains the same literal and its negation, mark the clause as unsatisfiable.
  - Otherwise, the clause can be satisfied. The valuation for each literal is F if they are negated, and T otherwise.

- If at least one clause is satisfiable, the entire problem is – and we can stop processing.

- If no clauses are satisfiable, the problem cannot be satisfied.

However, as the second formula we converted illustrates, there can be an *exponential increase* in size for an arbitrary propositional logic formula written in CNF. Hence, in the general case, we cannot use this conversion procedure to solve SAT in linear time for any Boolean logic formula.

2.3.2. Cook-Levin and NP Completeness

SAT is NP complete. It is computationally straightforward (i.e. possible in linear time) to check if a given valuation of variables is satisfying by evaluating the clauses within the entire formula. Furthermore, it has been shown that any problem in NP can be reduced to SAT by way of a nondeterministic Turing machine encoding (Cook, 1971) – this is the statement of the Cook-Levin theorem.

This has an impact on the practical application of SAT solvers and the reliability of such applications. Whilst heuristics can be used to efficiently solve some SAT problems, there is no guarantee that a particular problem will be able to be solved in a reasonable timeframe. Generally though, if a solver returns SAT/UNSAT instead of timing out, it can be assumed that the result is correct with a good degree of confidence.

2.3.3. Knuth's Algorithm A

First discussed in Knuth (2015), Knuth's *Algorithm A* (also known as SAT0) solves SAT by backtracking. Designed as a basic solution to the problem of designing a SAT solver, it is an improvement on the naïve algorithm discussed earlier. For a SAT problem involving n variables, the algorithm proceeds by setting the nth variable to its "most plausible" value and then recursively doing the same for the $n-1$ previous variables. If at any point a contradiction is encountered, the value assigned to n is flipped and the process repeats. For literals which are never negated in any clause, we can assume that they are true without issue. These are known as "pure literals", a concept which reappears in the DPLL algorithm discussed later.

At each stage, clauses must be modified to take into account the assignments, such as $x_1 = T$. This is achieved by removing any *clause* which contains a literal $l = x_1$. For any clauses containing $l = \neg x_1$, this literal must be removed because it can no longer be used to satisfy the clause now that x_1 has been set. If x_1 is set to F, the same procedure happens in reverse (clauses containing $l = \neg x_1$ are removed, literals requiring $l = x_1$ are removed).

2.3.4. DPLL

A more advanced approach than Knuth's SAT0 (discussed in section 2.3.3) is the *Davis–Putnam–Logemann–Loveland* (DPLL) algorithm. DPLL underpins a number of modern SAT solving software such as Microsoft's *Z3* where it powers the core theory solver (De Moura

and Bjørner, 2008). The algorithm was developed by improving on the DP algorithm which was published in 1960 in Davis and Putnam (1960).

For an arbitrary CNF SAT problem, the algorithm uses a process known as *unitary resolution*. If all previous variables in a CNF clause are false, the disjunctive nature implies that the last literal *must* be true; DPLL applies this process recursively (Russell and Norvig, 2016). As briefly mentioned in algorithm 2.3.3, DPLL also uses the concept of a "pure literal" which is defined as a literal for which its negation does not appear in the Boolean formula.

Two operations are used:

UnitPropagate If the clause contains *one* unassigned literal, set the valuation for the variable in order to satisfy it.

PureLiteralAssign Pure literals do not impact the search space, because they can always be satisfied with one truth value. This operation simplifies the clause using this fact.

The algorithm terminates in one of two cases [2], outlined below:

Either the CNF formula ϕ is found to comprise a consistent set of literals – that is, there is no l and $\neg l$ for any literal l in the formula. If this is the case, the variables can be trivially satisfied by setting them to the respective polarity of the encompassing literal in the valuation. Otherwise, when the formula contains an empty clause, the clause is vacuously false because a disjunction requires at least one member that is true. In this case, the existence of such a clause implies that the formula (evaluated as a *conjunction* of all clauses) therefore cannot evaluate to true and must be unsatisfiable.

The full pseudo-code for the algorithm is shown below. DPLL's input is a set of clauses ϕ and the algorithm returns a Boolean value representing whether the problem is satisfiable or not.

⚙ **Algorithm 2.1:** Pseudo-code for DPLL algorithm (Russell and Norvig, 2016)

function DPLL(ϕ)
    if IsConsistent(ϕ) **then** // *Clauses contain no x_1 and $\neg x_1$*
        return TRUE // *We were able to find a satisfying valuation*
    end if
    if HasEmpty(ϕ) **then**
        // *At least one clause in the formula is empty/unsatisfiable*
        return FALSE
    end if
    for unit clause $\{l\} \in \phi$ **do** // *Unit clauses containing one unassigned literal*
        $\phi = $ UnitPropagate(l, ϕ) // *Apply unit propagation operation*
        return DPLL(ϕ)
    end for
    for pure literal $l \in \phi$ **do** // *Pure literals are never negated*
        // *Delete pure literals, these do not impact search*

---

[2]These DPLL termination conditions are taken verbatim from an encyclopedic description written previously by the author in *Williams, A. (2019), 'DPLL algorithm (section: algorithm termination)'*.

```
    ϕ = PURELITERALASSIGN(l,  ϕ)
        return DPLL(ϕ)
    end for
    l = PICKLITERAL(ϕ)
    // Arbitrarily pick an unassigned literal
    // Try literal set to true, short circuit success, otherwise try false
    return DPLL(ϕ[T/l]) or DPLL(ϕ[F/l])
end function
```

Examples

The examples below show how DPLL proceeds for a satisfiable and unsatisfiable formula.

> **Example:** $(a \vee \neg b) \wedge (\neg a \vee c \vee b) \wedge (\neg c \vee \neg a)$

Recursive calls are shown with another level of indentation in the trace below.

⚙ DPLL invocation: symbols unassigned: $[a, b, c]$, current model $\{\}$

→ Consistency check failed, we are not done yet

→ No empty clauses yet, we can continue

→ There are no pure literals available yet

→ There are no unit clauses available yet

→ Attempt $a = $ true, call DPLL

   ⚙ DPLL invocation: symbols unassigned: $[b, c]$, current model $\{a = T\}$

   → Consistency check failed, we are not done yet

   → No empty clauses yet, we can continue

   → Apply PureLiteralAssign on b, setting value to true

      ⚙ DPLL invocation: symbols unassigned: $[c]$, current model $\{a = T, b = T\}$

      → Consistency check failed, we are not done yet

      → No empty clauses yet, we can continue

      → Apply PureLiteralAssign on c, setting value to false

         ⚙ DPLL invocation: symbols unassigned: $[\ ]$,
            current model $\{a = T, b = T, c = F\}$

         ✓ Consistency check passes, returning SAT and current
            model $\{a = T, b = T, c = F\}$

> **Example:** $(\neg b) \wedge (\vee c \vee b) \wedge (\neg c)$

Recursive calls are shown with another level of indentation in the trace below.

⚙ DPLL invocation: symbols unassigned: $[b, c]$, current model $\{\}$

→ Consistency check failed, we are not done yet

→ No empty clauses yet, we can continue

→ There are no pure literals available yet

→ Apply UnitPropagate on b, setting value to false

  ⚙ DPLL invocation: symbols unassigned: $[c]$, current model $\{b = F\}$

  → Consistency check failed, we are not done yet

  → No empty clauses yet, we can continue

  → There are no pure literals available yet

  → Apply UnitPropagate on c, setting value to true

    ⚙ DPLL invocation: symbols unassigned: $[\,]$,
      current model $\{b = F, c = T\}$

    → Consistency check failed, we are not done yet

    ✕ Every clause is empty, returning with UNSAT and current
      model $\{b = F, c = T\}$

DPLL is straightforward to implement and libraries exist for the JVM and .NET platforms. The code sample below shows how to work with the AIMA Java library built from the descriptions in Russell and Norvig (2016):

{ } Code Listing Using the DPLL library   Java

```java
// Set up the symbols and literals
DPLLSatisfiable dpllSatisfiable = new DPLLSatisfiable();
PropositionSymbol a = new PropositionSymbol("A");
PropositionSymbol b = new PropositionSymbol("B");
PropositionSymbol c = new PropositionSymbol("C");

ComplexSentence notA = new ComplexSentence(Connective.NOT, a);
ComplexSentence notC = new ComplexSentence(Connective.NOT, c);

// (a \lor \neg b) \land (\neg a \lor \neg b) \land c
Sentence problem = Sentence.newConjunction(
    Sentence.newDisjunction(a, b),
    Sentence.newDisjunction(c, notA),
    Sentence.newDisjunction(notA, notC)
);

Set<Clause> clauses = ConvertToConjunctionOfClauses.convert(
    problem
```

```java
).getClauses();
List<PropositionSymbol> result = new ArrayList<>(
    SymbolCollector.getSymbolsFrom(problem)
);

// Create empty model
Model model = new Model();
// Do the satisfiability check
boolean dpllResult = dpllSatisfiable.dpll(clauses, result, model);
System.out.println(dpllResult); // true = SAT, false = UNSAT
```

Although somewhat verbose, the library and its object-oriented representation of SAT formulae are intuitive to work with.

2.3.5. Satisfiability Modulo Theories (SMT)

Thus far, we have only considered solving problems involving Boolean variables and their negation. With SMT, we extend the problem to satisfiability of arbitrary first-order logic theories. This is more flexible and reduces the need for encoding work to formulate problems in terms of CNF formulae. With an SMT solver, such as Z3, we can solve systems of equations as shown in the code sample below:

{ } Code Listing Solving a simple multivariate equation with Z3   Python

```python
import z3

# define two variables, x and y, of 'sort' (type) Int
x = z3.Int("x")
y = z3.Int("y")

# solve the system of three equations and assign to ans
ans = z3.solve(x > 1, y > 3, y*y+x*x == 20)

# ans = [y = 4, x = 2]
```

Using the Python bindings for Z3, we solve the equation $y^2 + x^2 = 20$ for integer x and y. When executed, the solver yields the valuation $x = 2$, $y = 4$.

We define a theory T as a tuple (Σ_T, I_T) where Σ_T is the alphabet/list of function symbols within the theory T and I_T defines the *interpretations* or meaning in the theory. For example, consider the theory defining less-than and greater-than on positive integer numbers:

$$T = (\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, <, >, x, y\}, I_T)$$

Then $x < 4$ for $x = 2$ is T under I_T.

Using CNF as before, a simple problem under this theory could require the clauses $(x > y \vee x < 10) \wedge (y < 5) \wedge (4 > x)$ to be satisfied. Most SMT solvers would then provide a *model*, assigning possible satisfying values to the variables in the problem. In this example, $x = 4$, $y = 4$ would suffice.

2.3.6. DPLL(T): Extending DPLL to Arbitrary Theories

Initially described in Ganzinger et al. (2004) and presented in Russell and Norvig (2016), DPLL(T) extends the power of DPLL to a theory T (as defined above) using a repeated two-phase approach:

- First, all functions under the theory T in the problem are replaced with a Boolean variable. DPLL then proceeds as described earlier to find a satisfying valuation for the Boolean variables.

- Once a full set of valuations are available, a specialised *theory solver* attempts to determine if the valuations can be satisfied under the theory T. If these valuations yield logical constraints that are contradictory, the original formula ϕ is augmented with an additional clause to reflect the requirement that the two constraints cannot both be true.

- If the formula has been modified as the result of a contradiction, the SAT solver is executed again to retrieve a new valuation – assuming it is possible to backtrack.

- If at any stage it is not possible to backtrack and the functions required to be true contradict each other, the entire problem is unsatisfiable.

In DPLL(T), the theory solver is queried as an oracle and used each time the SAT solver has found a valuation. However, some heuristics used by the SAT solver cannot be used as-is within the context of an SMT problem. Barrett et al. notes that the *pure literal rule* must be disabled, because atoms within an SMT problem are not necessarily independent[3] as they are in a standard SAT problem (Barrett et al., 2002).

Below, we see how the DPLL(T) algorithm proceeds with an example within the context of the linear integer arithmetic theory:

> 🧩 **Example:** $((b > 7) \vee (a + 5 > 4) \wedge (b < 5 \vee \neg(b > 7)))$
>
> We begin by deriving a SAT formula for the problem: $(A \vee B) \wedge (C \vee \neg A)$.
>
> Perform DPLL:
>
> ⚙️ DPLL invocation: symbols unassigned: $[A, B, C]$, current model $\{\}$

---

[3]Observe that the truth value of a clause such as $A = \neg(x < 5)$ impacts the truth value of $B = (x < 10)$

→ Consistency check failed, we are not done yet

→ No empty clauses yet, we can continue

⊳⊳ Skipping pure literal check

→ There are no unit clauses available yet

→ Attempt A = true, call DPLL

    ⚙ DPLL invocation: symbols unassigned: $[B, C]$, current model $\{A = T\}$

    → Consistency check failed, we are not done yet

    → No empty clauses yet, we can continue

    ⊳⊳ Skipping pure literal check

    → Apply UnitPropagate on C, setting value to true

        ⚙ DPLL invocation: symbols unassigned: $[B]$, current model $\{A = T, C = T\}$

        ✓ Consistency check passes, returning SAT and current model $\{A = T, C = T\}$

Earlier, we assigned $A = (b > 7)$ and $C = (b < 5)$. The SAT solver has required that A be true. However, C contradicts this requirement by requiring $b < 5$ where $5 < 7$. The theory solver will now augment the original SAT problem with a requirement that A and C cannot both be true.

$$(A \lor B) \land (C \lor \neg A) \land \boxed{(\neg A \lor \neg C)}$$

We perform DPLL on the new SAT formula:

⚙ DPLL invocation: symbols unassigned: $[A, B, C]$, current model $\{\}$

→ Consistency check failed, we are not done yet

→ No empty clauses yet, we can continue

⊳⊳ Skipping pure literal check

→ There are no unit clauses available yet

→ Attempt A = true, call DPLL

    ⚙ DPLL invocation: symbols unassigned: $[B, C]$, current model $\{A = T\}$

    → Consistency check failed, we are not done yet

    → No empty clauses yet, we can continue

    ⊳⊳ Skipping pure literal check

    → Apply UnitPropagate on C, setting value to true

        ⚙ DPLL invocation: symbols unassigned: $[B]$, current model $\{A = T, C = T\}$

        → Consistency check failed, we are not done yet

✗ Every clause is empty, returning with UNSAT and current model
$\{A = T, C = T\}$

→ Attempt with $A =$ true failed, we have backtracked

→ Attempt $A =$ false

⚙ DPLL invocation: symbols unassigned: $[B, C]$, current model $\{A = F\}$

→ Consistency check failed, we are not done yet

→ No empty clauses yet, we can continue

⯈⯈ Skipping pure literal check

→ Apply UnitPropagate on B, setting value to true

⚙ DPLL invocation: symbols unassigned: $[C]$, current model $\{A = F, B = T\}$

✓ Consistency check passes, returning SAT and current model
$\{A = F, B = T\}$

We now have a satisfying valuation $\{A = F, B = T\}$. We query the theory solver, are $\neg A$ and B contradictory?

- $\neg A = \neg(b > 7) = (b \leq 7)$

- $B = (a + 5 > 4) = a > -1$

These concern different variables, so there is no problem with both of these atoms being true. As the model did not mention a truth value for C, the problem is satisfiable with either truth value for C.

An example model under this theory would be $\{b \to 4, a \to 1\}$.

Although we have discussed theories relating to linear integer inequalities, this approach is equally applicable to other theories. Z3str3, which powers string solving in Z3, is built on this principle and uses a theory solver that is equipped with knowledge of non-deterministic finite automata (Berzish et al., 2017).

2.4. Refinement Types

Within type systems, *refinement types* allow for predicate-based constraints to be applied to types in order to restrict the domain of elements which belong to the type. The concept of refinement types was first introduced in Freeman and Pfenning (1991)'s paper on *Refinements Types in ML* which "preserve type inference" whilst "allowing more errors to be detected at compile time". Refinement types have subsequently been developed for languages such as Haskell (in the form of *Liquid Haskell*, discussed in section 2.5.2), Scala and TypeScript.

A local variable used to store natural numbers could be constrained via a refinement type

such as $\{n : \mathbb{N} \mid n \leq 3\}$; only $\{0, 1, 2, 3\}$ would then be permitted for values of n under this constraint. Equally, we can apply the same concept to strings. A type with a length constraint could be written as $\{s : \Sigma^* \mid |s| = 4\}$ to specify a domain of "all strings which are 4 characters in length".

An intelligent type checker should be able to determine if values of one refinement type are compatible with another. For example, values in $\{n : \mathbb{N} \mid n \leq 2\}$ should be assignable to a member of type $\{n : \mathbb{N} \mid n \leq 5\}$.

In order to model common input validation practice, we allow regular expression membership to be expressed using refinement types. For example, $\{s : \Sigma^* \mid s \in L(ba+)\}$ would allow `"baa"` to be assigned as a value of some variable s, but not `"a"`. This refinement type can then be applied to local variables as well as function return types and parameter types.

The issue of compatibility arises for regular expressions, too. Intuitively, we know that strings in $L(a+)$ are also a member of $L(a*|b*)$.

2.4.1. Smart Constructors

As an alternative to refinement types, it is possible in some languages to define new types with so-called 'smart' constructors which enforce a range of constraints when creating an instance of the type. For example, the value could be checked to ensure it was greater than a certain number. Violations could return a `Nothing` type (or equivalent).

The example below from Clemens (2019) shows how one could implement a type to represent a reasonable human age in the Rust programming language:

{ } Code Listing An Age type in Rust   Rust

```rust
fn age() -> Age {
  // guaranteed at compile time to return an Age containing a u8 < 100
}

// an age is a struct containing an unsigned 8 bit int
struct Age(u8);

impl Age {
  // used to safely create an Age
  // will return an Option with an Age if the value is sensible
  fn new(age: u8) -> Option<Age> {
    if age < 100 {
      Some(Age(age)) // age is valid, return an Age
    } else {
      None // yield nothing
    }
  }
```

```rust
  // unwraps the age to return integer if underlying value required
  fn into_inner(self) -> u8 {
    self.0
  }
}

// allow the Age type to be used as if it were a u8
// by automatically dereferencing using the struct member
impl std::ops::Deref for Age {
  type Target = u8;

  fn deref(&self) -> &Self::Target {
    &self.0
  }
}
```

The idea is that other parts of the codebase can return/accept the `Age` type. This provides a compile-time guarantee that, if an `Age` type is present, its inner value meets the conditions defined within the smart constructor.

There are a few disadvantages to this approach. Firstly, the implementation is rather verbose. A new type needs to be created for each different constraint, and the various functions must be implemented alongside it. Refinement type systems make it much more straight-forward to define new refined types and determine if two types are compatible.

Additionally, the value check is carried out at runtime. In contrast, a type checker which supports refinement types is able to determine whether e.g. constants belong to the type *at compile-time*. As a result, no check needs to take place at runtime and the values can be "baked in" to the executable with reduced overhead.

2.5. Prior Art

There is a wealth of existing technical work within the areas of static analysis and program verification. This section aims to explore, and evaluate, some of the these works.

2.5.1. Formal Languages

Rex

Rex is a tool produced within the Research in Software Engineering (RiSE) research group within Microsoft. Veanes et al. introduce the concept of ϵ-SFAs (Symbolic Finite Automata) which provide a more flexible mapping to regular expressions (as they are implemented in modern programming languages) (Veanes et al., 2010).

Rex SFAs use formulae instead of characters to define transitions. Constructing automata for regular expressions involving character classes, or matching unicode characters (for which the overall alphabet size is large) is much more tractable using the SFA representation.

⊞ Figure 2.4: A simple SFA for the regex `[A-F]+$`

$$x \geq A \wedge x \leq F$$

start $\longrightarrow q_a \qquad q_b$

$$x \geq A \wedge x \leq F$$
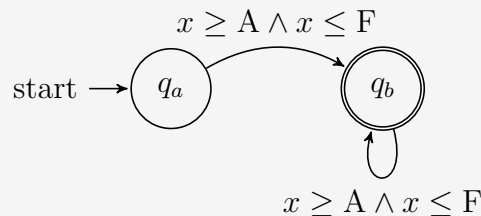
Microsoft have described a random walk algorithm that can operate on an SFA to generate valid string members that are matched by the regular expression (Veanes, 2013). This does not directly help us reduce false positives generated by static analysis tools, but it is an exciting development for offensive security. As we have discussed, many applications rely on regular expressions for initial validation logic. Using Rex as a *fuzzing* tool, a security researcher could generate input deemed valid by the validation logic which caused the system-under-test to crash or exhibit unintended functionality. This could also help to uncover bugs in sanitisers and filters such as those designed to prevent cross-site scripting by disallowing HTML.

2.5.2. Program Verification

Dafny

Dafny is a language designed by Microsoft Research with built-in static verification functionality and modern programming language features (including polymorphic types). The language itself is imperative and allows programmers to specify functions with pre and post-conditions which are verified at compile time (Leino, 2010). Recent versions of Dafny allow programs to be compiled into executables targeting the .NET framework (Leino and Moskal, 2014).

{ } Code Listing Factorial function and a pre-condition violation   Dafny

```
function Fac(n: int): int
requires n >= 1 // pre-condition
{
        // base case         recursive case
        if n == 1 then 1 else n * Fac(n - 1)
}

method Main()
```

```dafny
{
    print Fac(-2); // violation
}
```

In this sample, we define a function called `Fac` which calculates the factorial of a given input integer n. For example, `Fac(3)` yields $3 \times 2 \times 1 = 6$. Using the `requires` keyword, we specify a pre-condition on the function to ensure that any argument must satisfy $n \geq 1$. The `Main` function then calls `Fac` with a negative number, which causes a verification error at compile time.

The language uses an intermediate representation language known as *Boogie* which is able to derive constraints. These constraints are passed to the Z3 SMT solver to verify the user's program. Listing 2.5.2 shows an example Dafny program which causes a compile-time verification error.

Dafny has some limited `string` support. An example is shown in listing 2.5.2 with a function pre-condition on the length of a string:

{ } **Code Listing** Length pre-condition for a string   Dafny

```dafny
function method Strings(name: string): string
requires |name| == 4
{
   "Hello " + name
}

method Main()
{
  print Strings("Adam"); // passes compile
  print Strings("Jonathan"); // violation
}
```

Strings in Dafny are treated as equivalent to the type `seq<char>` (that is, a sequence of characters) so although it is possible to retrieve e.g. the length of such a sequence, there is no built-in regular expression support. Whilst it *is* possible to write a helper function and call it from within a pre-condition or post-condition, it then becomes necessary to manually implement the regular expression check in imperative code (as in listing 2.5.2).

{ } **Code Listing** Manually implementing a regular expression pre-condition   Dafny

```dafny
function method Test(a: string): string
requires StrTest(a)
{
    a + " was accepted"
}
```

```
function StrTest(n: string): bool
{
    // Manually implement the regex a+|g+
    //    For every character index c inside the array bounds of n
    //    check that the character at position c is 'a' or 'g'
    forall c: int :: 0 <= c < |n| ==> n[c] == 'a' || n[c] == 'g'
}

method Main()
{
    print Test("aaaaa"); // works, prints "aaaaa was accepted"
    print Test("Violating String"); // fails to compile
}
```

It should be immediately apparent how much more verbose this approach is than a refinement type. For more complex regular expressions, it becomes difficult to manually implement the DFA/NFA required to check the string which increases the likelihood of implementation bugs in the verification part of the program. This should be avoided at all costs, since it can cause false confidence in the correctness of the code.

There is some limited interoperability support with other .NET code (Wilcox, 2018), but the absence of a standard library or input/output support limits Dafny's potential to an existence as an interesting research language.

Liquid Haskell

Liquid Haskell is a program verifier for Haskell. Refinement types are supported in the form of special nested comments which specify *liquid types*.

The logic available for use in these type specifications is limited to ensure that the derived *verifiable conditions* that are eventually passed to the SMT solver are *decidable*. Thus, whilst Liquid Haskell supports basic logical expressions, relations and operators, there is no support for regular expression membership checks or arbitrary code execution to implement such functionality.

Listing 2.5.2 shows an example of a type constrained using Liquid Haskell to only permit odd integer members.

{ } Code Listing An Odd type to model odd numbers   Haskell

```
module Odds where

{-@ type Odd = {v:Int | v mod 2 = 1} @-}
```

```
{-@ oddNumbers :: [Odd] @-}
oddNumbers      = [7, 5, 1, 3, 667]

{-@ oddNumberThatIsActuallyEven :: Odd @-}
oddNumberThatIsActuallyEven = 8


---


oddNumberThatIsActuallyEven    :: Int
oddNumbers                     :: [Int]
```

A violation is triggered by **oddNumberThatIsActuallyEven** which, although a member of the **Odd** data type, has been assigned a value of 8.

2.5.3. Static Analysis and Security Tooling

Security Code Scan

Security Code Scan is a tool built to analyse C# programs using the *Roslyn* inspections system. The project continues the work of the earlier *Roslyn Security Guard* and includes a variety of different rules aiming to flag potential security issues arising from both user-input handling and oversights during the development process.

Support is included for taint analysis. Users can define their own taint types in a YAML configuration file, which are used in addition to a built-in database. These types can then be applied to different sources of data. By allowing multiple different taint types like this, Security Code Scan is much more flexible than many traditional taint-tracking systems which take a binary approach and label data as either user-input or benign. The use of unique types for validated/encoded input also helps to cut down on the number of false positives.

{ } Code Listing Specifying taint types which are applied to pieces of data   YAML

```
# Used in sanitisers and validators
TaintTypes:
  - LocalUrl # URL which we know is local (IsLocalUrl check performed)
  - HtmlEscaped # User input which we know is escaped
  - LdapFilterEncoded # LDAP filter string which we know is encoded
  - LdapDistinguishedNameEncoded # LDAP DN which has been encoded
  - AbsolutePath # Absolute path generated from a safe routing function
  - UrlEncoded # URL which has been through the encoding process
  - UrlPathEncoded # Path component of a URL which has been encoded
```

In the configuration above, a number of types are defined. Later on, we map one of these types to the return values of well-known library functions that safely encode user input for use in

LDAP filters. We set-up rules for both the sanitisers which make user input safe and the taint sinks which we want to ensure are used safely.

{ } Code Listing Rules relating to LDAP injection   YAML

```yaml
Behavior:
  Encoder_LdapFilterEncode:
    Namespace: Microsoft.Security.Application # Namespace of the class
    ClassName: Encoder # Enclosing class
    Name: LdapFilterEncode # When this method is called on the above class
    Method: # Method based rule
      Returns: # We consider its return value
        Taint: LdapFilterEncoded # And taint return value as LdapFilterEncoded
        TaintFromArguments: [0]
  DirectorySearcher_Filter:
    Namespace: System.DirectoryServices # Namespace of the class
    ClassName: DirectorySearcher  # Enclosing class
    Name: Filter # Name of the property
    Field: # Field/property based rule
      # To be safe, this field's value must be LdapFilterEncoded
      # SCS0031 is the name of the inspection
      Injectable: [SCS0031: LdapFilterEncoded]
```

Whilst this approach is quite flexible, the tool is unable to determine if the *form* of user input inherently makes exploitation impossible.

For instance, an ID validated to be purely numeric that is later passed to an LDAP filter is unlikely to leave much scope to exploit LDAP injection–even if the input has not been correctly passed to a filter encoding method in line with security best practice.

{ } Code Listing Rules relating to LDAP injection   C#

```csharp
public string GetSurnameForUser(string username)
{
    var directoryEntry = new DirectoryEntry("LDAP://acme.corp");
    var searcher = new DirectorySearcher(directoryEntry);
    searcher.Filter = "(cn=" + username + ")";
    var result = searcher.FindOne();
    return result.Properties["sn"][0].ToString();
}
```

Snyk

Snyk is a polyglot tool which analyses system dependencies to find components with known security vulnerabilities. Build files which specify dependencies are parsed and analysed. An example for the *Gradle* build system, which is popular amongst JVM developers, is shown below for an Android application:

{ } Code Listing Specifying dependencies using a build tool   Gradle
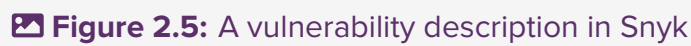
```
dependencies {
    testImplementation 'junit:junit:4.12'
    testImplementation 'org.mockito:mockito-core:1.10.19'
    implementation 'com.android.support:appcompat-v7:27.1.1'
    implementation 'com.android.support:design:27.1.1'
    implementation 'com.android.support:customtabs:27.1.1'
    implementation 'com.android.support:cardview-v7:27.1.1'
    implementation 'com.crashlytics.sdk.android:crashlytics:2.9.5'
}
```

Figure 2.5 is an illustrative vulnerability report for the *My Warwick* Android application. The issue claims to exist in a Java utility library, *Guava*. The nature of deserialisation vulnerabilities is that an application must:

- Accept user input for deserialisation

- Unsafely deserialise the user input, in a manner which places no restriction on the types the attacker can deserialise to

- Have, on its classpath, a class which can be used by an attacker as a *gadget* to execute code or carry out a denial of service (DoS) attack

Snyk vulnerabilities relating to deserialisation tend to be raised in large volumes for each *gadget* discovered in a library. The system gives no weight to whether the application deserialises user input, does so in an unsafe way which allows any type to be used, or indeed does any deserialisation at all. In *My Warwick*'s case, no automatic deserialisation is present at all, and so the is completely unexploitable. This problem is a consequence of Snyk only analysing dependencies and not performing static analysis on the codebase to determine if the problem is relevant or not.

At the time of writing, the company has introduced "runtime monitoring" to determine if insecure functions are actually called. This functionality increases the value of the vulnerability reports, but only supports a limited number of platforms and requires running an agent alongside the application to collect and send runtime data.

Figure 2.5: A vulnerability description in Snyk

MEDIUM SEVERITY

🛡 Deserialization of Untrusted Data

Vulnerable module: com.google.guava:guava
Introduced through: com.google.guava:guava@24.0-android

Detailed paths

- **Introduced through**: project@0.0.0 › com.google.guava:guava@24.0-android

  Remediation: Upgrade to com.google.guava:guava@24.1.1-jre.

Vulnerable functions

`com/google/common/util/concurrent/AtomicDoubleArray.readObject()`

Overview

com.google.guava:guava is a set of core libraries that includes new collection types (such as multimap and multiset,immutable collections, a graph library, functional types, an in-memory cache and more.

Affected versions of this package are vulnerable to Deserialization of Untrusted Data.

During deserialization, two Guava classes accept a caller-specified size parameter and eagerly allocate an array of that size:

- `AtomicDoubleArray` (when serialized with Java serialization)
- `CompoundOrdering` (when serialized with GWT serialization)

An attacker may be able to send a specially crafted request which with then cause the server to allocate all it's memory, without validation whether the data size is reasonable.

More about this issue   📌 Create a Jira issue UPGRADE   👁 Ignore

Design and Implementation

This chapter discusses the design decisions made throughout the project and the reasoning behind them. It additionally covers how the implementation was approached, and the tools and libraries used to produce the prototype.

3.1. Design

The need for our language to be predictable and familiar to developers is clear. We therefore present an imperative proof-of-concept language, with a C-like syntax. The aim is that developers, and particularly those that work with JVM technologies, will be able to write critical sections of code within our language, where regex refinements are available.

This section covers the design of the language's syntax, its type system, the foreign function interface (FFI) and interpreter. At build time, the language tooling will progress through a series of stages:

☰ Parsing ⟩ ⊶ AST Construction ⟩ ☑ Type checking ⟩ Reporting ⟩ ⚙ Interpreting ⟩

3.1.1. Syntax

The C-like syntax, with some inspiration from other languages such as Go, is augmented with the use of square brackets to specify refinement types. Below, we show a high-level BNF grammar for our language. This was constructed, made unambiguous and then used as a basis during the implementation phase.

We begin by defining our main rule to represent a complete program. Programs contain a number of functions, which are declared with an identifier, number of arguments and a return type.

$\langle program \rangle$ ::= $(\langle function \rangle$ \n)* $\langle function \rangle$ ::= $\langle function\text{-}signature \rangle$ $\langle body \rangle$ '}'

$\langle function\text{-}signature \rangle$ ::= 'function' ' ' $\langle identifier \rangle$ '(' $\langle parameter\text{-}decl \rangle$ '): ' $\langle type \rangle$ '{\n'

$\langle body \rangle$ ::= $(\langle body\text{-}line \rangle$ \n)*

$\langle parameter\text{-}decl \rangle$ ::= $\langle expr \rangle$ ': ' $\langle type \rangle$
                             | $\langle expr \rangle$ ': ' $\langle type \rangle$ ', ' $\langle parameter\text{-}decl \rangle$
                             | $\langle empty \rangle$

We can now start to define what is permitted inside the lines that sit within a function body. We want to be able to declare local variables, assign values to them, call functions (discarding any return type) or control the flow of execution using *selection.*

⟨body-line⟩	::=	*⟨var-assignment⟩*
	|	*⟨return-stmt⟩*
	|	*⟨var-decl⟩*
	|	*⟨function-call⟩*
	|	*⟨if-stmt⟩*

⟨var-assignment⟩	::=	*⟨identifier⟩* '`=`' *⟨expr⟩*

⟨return-stmt⟩	::=	'`return`' *⟨expr⟩*

⟨var-decl⟩	::=	'`var`' *⟨identifier⟩* '`: `' *⟨type⟩*

⟨function-call⟩	::=	*⟨identifier⟩* '`(`' *⟨argument-list⟩* '`)`'

⟨argument-list⟩	::=	*⟨expr⟩* | *⟨expr⟩* '`,`' *⟨argument-list⟩*
	|	*⟨empty⟩*

Selection in the language is implemented with an **if-else if-else** construct as one might see in other imperative languages such as C. Such a statement has no limit on the number of **else if** components.

⟨if-stmt⟩	::=	'`if`' ' ' '`(`' *⟨expr⟩* '`)`' ' ' '`{ \n`' *⟨body⟩* '`}`' *⟨optional-else⟩*

⟨optional-else⟩	::=	' ' '`else`' ' ' '`{ \n`' *⟨body⟩* '`}`'
	|	' ' '`else`' ' ' if_stmt
	|	*⟨empty⟩*

We now turn our attention to the type system, and support for refinements (encompassed in the `<constraint>` rule) in the grammar. The language supports the following type keywords:

Type Keyword	Description
`string`	A collection of characters. May be refined to be a regular expression member.
`uint`	An unsigned integer. May include a refinement to be greater/less than a threshold.
`bool`	An ordinary Boolean value which may be `true` or `false`. No applicable refinements.
`void`	Used for functions with no return type.

We define the `<type>` class to represent a type reference. This class is then used throughout the grammar for function parameter types, local variables types and return types.

$\langle type \rangle$ ::= $\langle type\text{-}keyword \rangle$ | $\langle type\text{-}keyword \rangle$ '`[`' $\langle constraint \rangle$ '`]`'

$\langle type\text{-}keyword \rangle$ ::= '`void`' | '`uint`' | '`bool`' | '`string`'

$\langle constraint \rangle$ ::= '`>`' $\langle number \rangle$
                     | '`<`' $\langle number \rangle$
                     | '`>=`' $\langle number \rangle$
                     | '`<=`' $\langle number \rangle$
                     | '`/`' $\langle re \rangle$ '`/`'

Integer refinements are a relatively straightforward affair, as shown above. We now parse regular expression refinements using the parsing hierarchy proposed in Cameron (1999). We amend the production rules used by Cameron to ensure they are unambiguous, by creating prime-suffixed classes with ϵ-productions.

$\langle re \rangle$ ::= $\langle simple\text{-}re \rangle$ $\langle union\text{-}prime \rangle$

$\langle union\text{-}prime \rangle$ ::= '`|`' $\langle re \rangle$ | $\langle empty \rangle$

$\langle simple\text{-}re \rangle$ ::= $\langle basic\text{-}re \rangle$ $\langle concat\text{-}prime \rangle$

$\langle concat\text{-}prime \rangle$ ::= $\langle simple\text{-}re \rangle$ | $\langle empty \rangle$

$\langle basic\text{-}re \rangle$ ::= $\langle kleene\text{-}star \rangle$ | $\langle plus \rangle$ | $\langle elementary\text{-}re \rangle$

$\langle kleene\text{-}star \rangle$ ::= $\langle elementary\text{-}re \rangle$ '`*`'

$\langle plus \rangle$ ::= $\langle elementary\text{-}re \rangle$ '`+`'

$\langle elementary\text{-}re \rangle$ ::= $\langle group \rangle$ | '`.`' | $\langle character \rangle$ | $\langle range\text{-}re \rangle$

$\langle group \rangle$ ::= '`(`' re '`)`'

$\langle range\text{-}re \rangle$ ::= $\langle positive\text{-}range \rangle$ | $\langle negated\text{-}range \rangle$

$\langle positive\text{-}range \rangle$ ::= '`[`' $\langle range\text{-}items \rangle$ '`]`'

⟨*negated-range*⟩   ::= '[^' ⟨*range-items*⟩ ']'

⟨*range-items*⟩   ::= ⟨*range-item*⟩ | ⟨*range-item*⟩ ⟨*range-items*⟩

⟨*range-item*⟩   ::= ⟨*character*⟩ '-' ⟨*character*⟩ | ⟨*character*⟩

We define expressions to permit basic arithmetic operations, nesting and function calls:

⟨*expr*⟩   ::= ⟨*expr*⟩ ('*'|'/') ⟨*expr*⟩
        | ⟨*expr*⟩ ('+'|'-') ⟨*expr*⟩
        | ⟨*value_ref*⟩
        | ⟨*expr*⟩ ' ' ('<'|'>'|'<='|'>='|'==') ' ' ⟨*expr*⟩
        | ⟨*function-call*⟩
        | ⟨*value-ref*⟩

⟨*value-ref*⟩   ::= ⟨*number*⟩ | ⟨*string-literal*⟩ | 'true' | 'false' | ⟨*identifier*⟩

Finally, we set-up the most basic rules for setting what is permitted as an identfier name, string literal or number:

⟨*number*⟩   :== ⟨*digit*⟩ | ⟨*integer*⟩ ⟨*digit*⟩

⟨*digit*⟩   :== '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

⟨*string-literal*⟩   :== '"{}' (⟨*character*⟩ *) '"{}'

⟨*identifier*⟩   :== ⟨*alpha*⟩ (⟨*alphanumeric*⟩ *)

The result is a small imperative language which can be used to implement simple logic. Code can be reused by way of user-defined functions. Included below are some illustrative examples of basic programs written in the discussed syntax:

{ } Code Listing A factorial function written in the RRT language   RRT

```
function Factorial(n: uint): uint {
    if (n == 0) {
        return 1
    }
    return n*Factorial(n-1)
}

function Main(): uint {
    return Factorial(4)
}
```

In this example, we use the unrefined `uint` type to implement the factorial function, where $5! = 5 \times 4 \times 3 \times 2 \times 1$.

We can also begin to use refinement types in our program code:

> **{ } Code Listing** A product function, which multiplies its arguments   RRT
>
> ```
> function Product(a: uint[<10], b: uint[<=15]): uint {
>     return a*b
> }
>
> function Main(): uint {
>     // prints 60
>     return Product(4, 15)
> }
> ```
>
> Here, we use integer refinement types to restrict the domain of the arguments to the `Product` function. The first number, `a`, is required to be less than 10 whereas `b` must be less than or equal to 15.

3.1.2. Abstract Syntax Tree

Combined with a parser generator, the grammar discussed in section 3.1.1 enables programs to be parsed into a data structure known as a *Concrete Syntax Tree* (CST) which maps directly to the grammar.

Whilst the CST is useful, most type checking will require a richer representation of a program written in our language. For example, the current grammar would parse refinements such as `uint[/.*/]`, To check for these issues and interpret code, we use an *Abstract Syntax Tree* (AST) representation of the program.

We designed the structure of the AST prior to implementing the system. Unlike some (generally more functional languages), we make a distinction between *statements* and *expressions*. The former do not necessarily yield a value in the same way as expressions. For example, an if statement may not return a value–it could simply call a function and rely on its side effects. We make no claim that such a language is more elegant than others, but adoption is necessary for security tooling to be effective; designing an imperative language which would be comfortable to e.g. Java developers was important, since they are responsible for a great proportion of production code.

We rely on the object-oriented concept of inheritance when modelling our AST entities. Expressions and statements can inherit from a common abstract base class and reuse their logic.

We first discuss the different types of expressions in our AST representation. There are 9 of these. We make a distinction between "value" expressions, where a value can be derived

immediately, and those which require computation to derive a value (e.g. a function call or arithmetic operation).

The diagram below show how the different expression types relate to each other:

▣ Figure 3.1: Expression AST objects
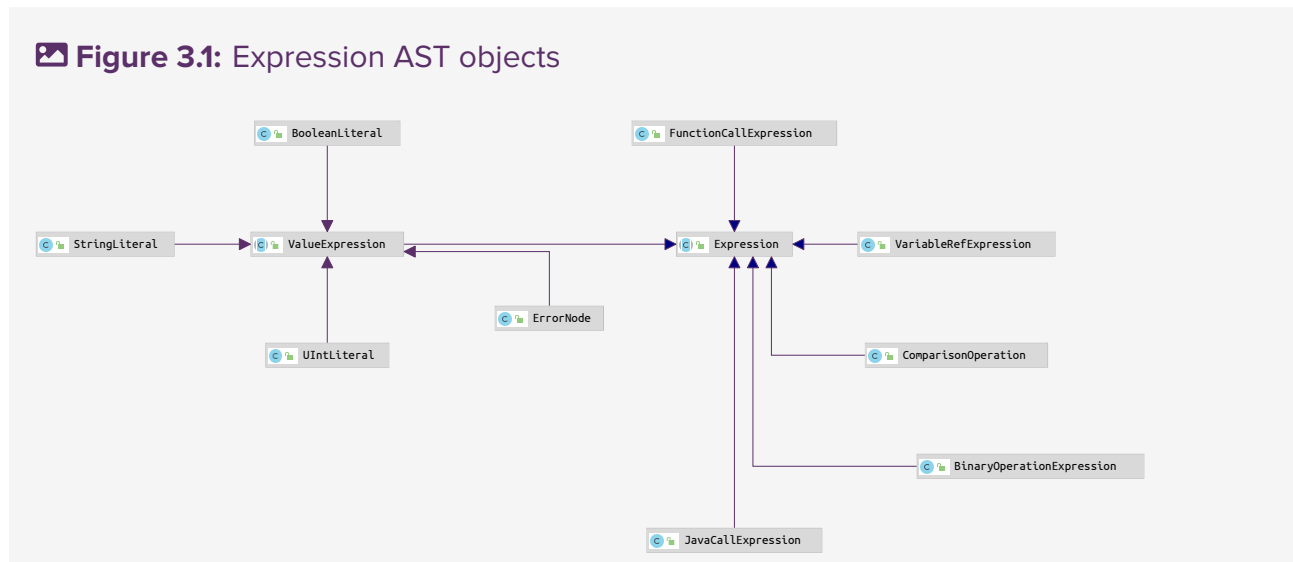
JavaCallExpression Models a expression which involves a foreign function interface call to a static JVM class method.

BinaryOperationExpression Represents an expression involving a binary operation, and two child nodes to which the operation is applied.

ValueExpression Parent object for value expressions, as discussed above.

FunctionCallExpression Represents a call to a non-void function, where the return value is used.

VariableRefExpression Encapsulates a reference to another variable which is in scope (either a function parameter or local variable).

Expression Parent of all expressions, defines operations to evaluate their value for use by the interpreter.

StringLiteral Represents a string literal defined in code, such as `"Hello, world"`.

UIntLiteral Represents an unsigned integer literal defined in code, such as `6`.
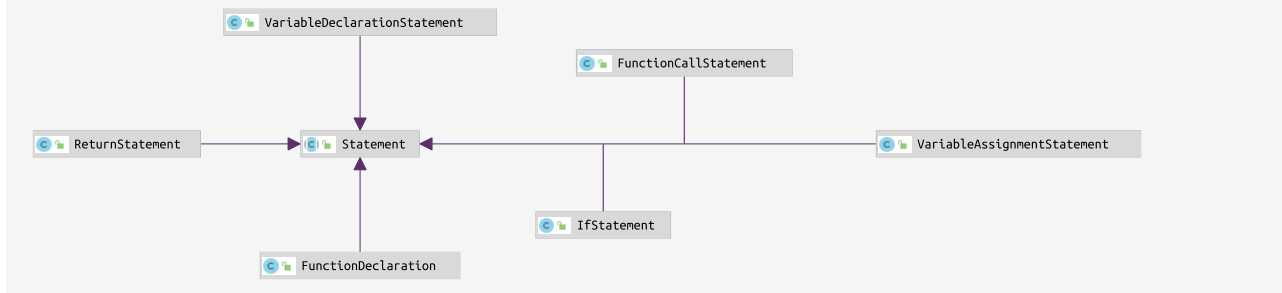
BooleanLiteral Represents a boolean literal–**true** or **false**.

All expression types must provide a way for the expression to be evaluated (yielding a value).

The diagram below shows the structure of our statement-related entities:

> **Figure 3.2:** Statement AST objects
>
>

VariableDeclarationStatement Models a statement of type `var variable: string[/A+/]`.

FunctionCallStatement Models a statement of type `Function()`, where the return value is unused.

VariableAssignmentStatement Handles assignment of a value to a variable, `variable="AAA"`

IfStatement Represents a control flow statement comprising an `if` clause and one or more `else`/`else if` clauses.

FunctionDeclaration Encapsulates data relating to a function such as the parameters it takes, an identifier and its return type.

ReturnStatement Represents a line in a function body which returns a value, e.g. `return 6`.

Each statement is endowed with an `execute` operation which can optionally return a value in the form of an expression.

Entities encapsulate data according to their purpose. An *IfStatement* object contains an expression that is evaluated to produce a Boolean value, a reference to the bodies of code to execute if the expression is true (or false) and a reference to a further *IfStatement* if an `else if` is present.

3.1.3. Type Checking

Types need to be checked in a number of situations:

- New value is assigned to a local variable

- Value is passed to a function as an argument

- Value is returned from a function

For simple types with no refinements, this is generally a straightforward check. As our language is strongly typed, we simply need to check that the expected and actual types are an exact match. No implicit conversions are supported in our language.

Refined types are more involved: it is not enough for us to check that the refinements are the same. Observe that a variable of type `uint[< 10]` is perfectly valid to return from a function with a return type of `uint[<= 20]`. We must instead determine if the refinement types are *compatible*. We define two refinement types τ_e, τ_a (expected and actual) to be compatible if there is no value $v : \tau_a$ in the actual type that is not in τ_e and is therefore in the complement of the expected type $(\tau_e)^C$.

A concrete example involving integer refinements is given below:

{ } Code Listing foo　　　　　　　　　　　　　　　　　　　　　　　　　RRT

```
function Demonstration(): uint[> 10] {
    var x: uint[> 5]
    x = 6
    return x
}
```

In the above listing, we need to compare types on 2 occasions:

1. When 6 is assigned to x (to check it is greater than 5)

2. When x is returned from the function `Demonstration` (to check > 5 is compatible with > 10)

The first check is trivially accomplished because there are no unknowns. The second check can be expressed in CNF as an SMT formula, using the complement as discussed: $x > 5 \wedge x <= 10$.

This problem can be checked using DPLL(T) as described in section 2.3.6. We formulate a SAT problem $A \wedge B$, receive a satisfying valuation $\{A = T, B = T\}$ and pass this to the theory solver. As the reader will appreciate, it is then trivial to satisfy this formula with any value between 6 and 10. Hence, we have found a violation and the types are incompatible.

The language type checker implements exactly this process. Constraints are built based on the types identified during AST construction and then solved using an SMT solver. We assume a theory solver is available for integer arithmetic and strings.

3.2. Implementation

3.2.1. Parsing

Introduction

In order to lex and parse user input into the AST structure discussed earlier, we use the *ANTLR4* parser generator. ANTLR4 generates $LL(*)$ parsers in a variety of languages, including Java, C# and Go (Parr and Fisher, 2011). These parsers are not restricted to a fixed number of lookahead tokens.

The grammar is split into two distinct files. One file defines the rules for the lexer, which is responsible for tokenising the source code. The second file describes the parser which ANTLR will generate. The parsing rules generally resemble the BNF discussed previously in section 3.1.1, so we will not dwell on them here. The lexer is more involved than a simple translation from BNF, because the language syntax requires that we make use of *lexical states*.

Consider the token definitions for variable/function identifiers and characters that can appear in a regular expression:

```
IDENTIFIER : [A-Za-z_] [A-Za-z_0-9]* ;
CHARACTER  : ~('\n'|'\r'|'.'|'('|')'|'['|']'|'*'|'+'|'/'|'|') | ESCAPED_META;
```

As the two token definitions overlap, they cannot appear in use together in the same lexical state. If they did, only one would be identified by the lexer, breaking the parsing of either the regular expressions, or the function definitions.

To solve this, we define two new lexical states using the **mode** keyword:

{ } **Code Listing** Using lexical states in the lexer   ANTLR

```
mode REGEX;
BEGIN_RE_RANGE : '[' -> pushMode(REGEX_RANGE) ;
END_RE_RANGE   : ']' ;
CHARACTER      : ~('\n'|'\r'|'.'|'('|')'|'['|']'|'*'|'+'|'/'|'|') | ESCAPED_META;
BEGIN_RE_GROUP : '(' ;
END_RE_GROUP   : ')' ;
MINUS          : '-' ;
DOT            : '.' ;
STAR           : '*' ;
PLUS           : '+' ;
ALTERNATION    : '|' ;
RANGE_NEGATE   : '^' ;
RE_DELIMITER_CLOSE: '/' -> popMode ;
META_CHAR      : (DOT|PLUS|STAR|ALTERNATION|'['|']'|'/') ;
fragment ESCAPED_FWD_SLASH: '\\/' ;
fragment ESCAPED_META: '\\' META_CHAR ;

mode CONSTRAINT;
GT: '>' ;
GE: '>=' ;
LT: '<' ;
```

```
LE: '<=' ;
EQ: '=' ;
CONSTRAINT_UINT: [0-9]+ ;
CONSTRAINT_SPACE: ' ' ;
RE_DELIMITER_OPEN: '/' -> pushMode(REGEX) ;
END_CONSTRAINT   : ']' -> popMode ;
```

When the lexer identifies characters which begin constraints after a type keyword, it is directed to enter the CONSTRAINT state. From here, the forward slash regular expression delimiter prompts the lexer to enter the REGEX state where regular expressions are tokenised. States are maintained in a manner not dissimilar to a call stack. If this delimiter is not observed, integer constraints can be tokenised instead.

Once the lexer reaches the end of an expression, it pops back into the previous lexical state and continues parsing.

Working with ANTLR

Once the grammar files are complete, ANTLR can be configured to generate parsers for a variety of languages. We decided to implement the type checker and interpreter in Java, so we target Java for the generated parser.

Using the *Gradle* build system, there is an official antlr plugin which provides a generateGrammarSource build task. When the grammar is modified, Gradle will regenerate the parser and lexer before compiling the rest of the Java code within the project.

Following parser and lexer generation, it is straightforward to use them to process user input:

{ } Code Listing Using ANTLR's lexing and parsing APIs   Java

```java
//Program to be parsed
String program = "function Main()\n{\n......";
// Lexer for the language, pointed at a stream constructed
// from the string above
PocLex lexer = new PocLex(CharStreams.fromString(program));
CommonTokenStream tokens = new CommonTokenStream(lexer);
// Parser, depends on tokens lexed by lexer
PocLang parser = new PocLang(tokens);
// Do the parsing, starting with the "program" non-terminal
ProgramContext parsedProgram = parser.program();
```

Implementing the *visitor* pattern (Gamma et al., 1995), we can then process the concrete syntax tree which ANTLR has generated by parsing the program.

```java
public class VisitorListener extends PocLangBaseListener {

    @Override
    public void enterProgram(PocLang.ProgramContext ctx) {
        // called when program non-terminal begins
    }
}
```

3.2.2. Integrations and Frontends

In this section, we present the various tools and integrations we have provide on top of the core parser, type checker and interpreter.

Command Line Interface

Initially, we built a basic command line interface which would read programs from the standard inpuit (stdin) stream.

```
→ PocLang (git:master) cat input.txt | ./gradlew run
> Task :run
Reading program from stdin (use Ctrl+D when finished)...
Violation via value "gggg"
L5:4 Return type string [(str.in.re x (re.union (re.+ (str.to.re "g")) (re.+
(str.to.re "f"))))] of function SecondaryFunction didn't satisfy string
[(str.in.re x (re.+ (str.to.re "f")))]
```

Web Application

> 💡 **TODO:** Screenshot, why it was developed

Rise4fun

Within Microsoft Research, the *Research in Software Engineering* (RiSE) group has built a tool known as *rise4fun*. This web based platform is an interactive sandbox where allows users interested in programming language research can experiment with software engineering tools such as Z3, Dafny and many others. Users can enter code in a syntax-highlighted editor and execute type-checkers and verifiers in their browser. Results are displayed in real-time.

We natively integrate with Microsoft's Rise4fun platform to showcase our language's features and allow users to experiment with refinement types. Much of the web application backend could be re-used, illustrating the flexibility of the parsing and type-checking APIs.

Figure 3.3: Example of a type violation, displayed in the Rise4fun environment

Testing

> 💡 **TODO:** Unit testing, manual testing (boundary conditions, adversarial input), integration testing (lexer, parser, interpreter)

Legal, social, ethical and professional issues

Initially covered in appendix A, we ensured that potential legal and ethical issues were minimised during the lifetime of the project. Much of the typical issues are not applicable to this project; no surveys were undertaken and no research was conducted which necessitated volunteers providing data.

However, the core motivations underpinning the project objectives lie in the realm of information security. Therein lies the biggest potential for ethical questions to be raised (Dark et al., 2008). Although our work on refinement types is designed primarily to assist the developer by providing assurance about the structure of flowing data, the nature of many security tools is that they can be used offensively.

For example, fuzzing tools which generate test data and are designed to help highlight memory corruption bugs, can be used by third-parties other than the original developers to audit software. Similarly, it is possible that the tooling discussed in this report could be used to discover potential exploitation opportunities in existing software, perhaps by porting logic into our language and adding refinement types in order to detect e.g. injection vulnerabilities by running the type-checker. SMT solvers such as Z3 combined with systems like Rex could be used to find bugs in sanitiser libraries designed to prevent e.g. XSS vulnerabilities.

Harper et al. discuss the ethics of security research in Harper et al. (2018). There is general consensus that "offline" security research into software[1] is ethically preferable to testing against real-world systems without authorisation, even if intentions are good and would fall into activity typical of a "grey hat" researcher. However, opinions differ on how discovered vulnerabilities should be disclosed. Schneier argues for *full disclosure*–where vulnerabilities are publicly disclosed–describing it as a "damned good idea" and citing the principle that "secrecy prevents people from accurately assessing their own risk" (Schneier, 2007). The Full Disclosure mailing list is a common avenue for disclosing vulnerabilities in this manner, with Rose arguing that other approaches are simply "security through obscurity":

> *We don't believe in security by obscurity, and as far as we know, full disclosure is the only way to ensure that everyone, not just the insiders, have access to the information we need.*

– Leonard Rose (creator of original *Full Disclosure* mailing list)

---

[1]By offline, we mean to capture research which does not involve testing against a third-party production environment. For example, local fuzzing of the OpenSSL cryptography library would fall into this category.

Evaluation and Conclusions   CHAPTER 6

6.1. Evaluation Against Existing Works

> 💡 **TODO:** Security Code Scan

6.2. Conclusions

We have designed and implemented a proof-of-concept language to support regular expression refinement types, demonstrating that it is feasible to implement type checking for such a language using SMT theory and a modern SMT solver.

Using a series of test cases, we have shown that regular expression membership violations are correctly detected and reported in a measure of time suitable for integrating into the build process.

Future Work

CHAPTER **7**

> 🔆 **TODO:** Flesh this out:
>
> - Non-regular "regular expressions" like we see in PHP, Perl etc
>
> - Integration with languages such as C#–take advantage of Roslyn? Scala/Java could also work by using annotations.
>
> - Extend Dafny/Haskell to take advantage of Z3str3?
>
> - Or, continue working on the proof of concept language to make it more palatable for serious use (more types, generics, .. objects)

Bibliography

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

Barrett, C. W., Dill, D. L. and Stump, A. (2002), Checking satisfiability of first-order formulas by incremental translation to sat, *in* E. Brinksma and K. G. Larsen, eds, 'Computer Aided Verification', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 236–249.

Berzish, M., Zheng, Y. and Ganesh, V. (2017), 'Z3str3: A string solver with theory-aware branching', *arXiv preprint arXiv:1704.07935* .

Boulianne, S. (2015), 'Social media use and participation: A meta-analysis of current research', *Information, communication & society* **18**(5), 524–538.

Cameron, R. D. (1999), 'Perl style regular expressions in prolog'.
URL: *http://www.cs.sfu.ca/c̃ameron/Teaching/384/99-3/regexp-plg.html*

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

Clemens, K. (2019), 'Smart constructors in Rust', IRC conversation, EsperNet.

Cook, S. A. (1971), The complexity of theorem-proving procedures, *in* 'Proceedings of the Third Annual ACM Symposium on Theory of Computing', STOC '71, ACM, New York, NY, USA, pp. 151–158.
URL: *http://doi.acm.org/10.1145/800157.805047*

Dark, M., Harter, N., Morales, L. and Garcia, M. A. (2008), 'An information security ethics education model', *Journal of Computing Sciences in Colleges* **23**(6), 82–88.

Davis, M. and Putnam, H. (1960), 'A computing procedure for quantification theory', *Journal of the ACM (JACM)* **7**(3), 201–215.
URL: *http://doi.acm.org/10.1145/321033.321034*

De Moura, L. and Bjørner, N. (2008), Z3: An efficient SMT solver, *in* 'International conference on Tools and Algorithms for the Construction and Analysis of Systems', Springer, pp. 337–340.

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.

Fox, M. L. (2010), 'Directgov 2010 and beyond: Revolution not evolution', *Letter to Francis Maude, Gov. uk* .

Freeman, T. and Pfenning, F. (1991), 'Refinement types for ML', *ACM Sigplan Notices* **26**(6), 268–277.
URL: *http://doi.acm.org/10.1145/113446.113468*

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995), 'Design patterns: Elements of reusable object-oriented software addison-wesley', *Reading, MA* p. 1995.

Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A. and Tinelli, C. (2004), DPLL (T): Fast decision procedures, *in* 'International Conference on Computer Aided Verification', Springer, pp. 175–188.

Harper, A., Harris, S., Ness, J., Eagle, C., Lenkey, G. and Williams, T. (2018), *Gray hat hacking: the ethical hacker's handbook*, McGraw-Hill Education.

Jayawardhena, C. and Foley, P. (2000), 'Changes in the banking sector–the case of internet banking in the uk', *Internet research* **10**(1), 19–31.

Kesselheim, A. S., Cresswell, K., Phansalkar, S., Bates, D. W. and Sheikh, A. (2011), 'Clinical decision support systems could be modified to reduce 'alert fatigue' while still minimizing the risk of litigation', *Health affairs* **30**(12), 2310–2317.

Knuth, D. E. (2015), *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*, 1st edn, Addison-Wesley Professional.

Leino, K. R. M. (2010), Dafny: An automatic program verifier for functional correctness, *in* E. M. Clarke and A. Voronkov, eds, 'Logic for Programming, Artificial Intelligence, and Reasoning', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 348–370.

Leino, K. R. M. and Moskal, M. (2014), Co-induction simply, *in* C. Jones, P. Pihlajasaari and J. Sun, eds, 'FM 2014: Formal Methods', Springer International Publishing, Cham, pp. 382–398.

Miltersen, P. B., Radhakrishnan, J. and Wegener, I. (2005), 'On converting CNF to DNF', *Theoretical computer science* **347**(1-2), 325–335.

OWASP (2017), 'The ten most critical web application security risks'.

Parr, T. and Fisher, K. (2011), 'LL (*): the foundation of the ANTLR parser generator', *ACM Sigplan Notices* **46**(6), 425–436.

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

Rabin, M. O. and Scott, D. (1959), 'Finite automata and their decision problems', *IBM journal of research and development* **3**(2), 114–125.

Russell, S. J. and Norvig, P. (2016), *Artificial intelligence: a modern approach*, Pearson Education Limited,.

Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L. and Jaspan, C. (2018), 'Lessons from building static analysis tools at google', *Communications of the ACM (CACM)* **61 Issue 4**, 58–66.
URL: *https://dl.acm.org/citation.cfm?id=3188720*

Schneier, B. (2007), 'Schneier: Full disclosure of security vulnerabilities a 'damned good idea'', *CSO Online* .

Schneier, B. (2011), *Secrets and lies: digital security in a networked world*, John Wiley & Sons.

Sipser, M. (2012), *Introduction to the Theory of Computation*, Cengage Learning.

Stuttard, D. and Pinto, M. (2011), *The web application hacker's handbook: Finding and exploiting security flaws*, John Wiley & Sons.

Veanes, M. (2013), Applications of symbolic finite automata, *in* 'CIAA'13', Vol. 7982 of *LNCS*, Springer, pp. 16–23.
  URL: *https://www.microsoft.com/en-us/research/publication/applications-of-symbolic-finite-automata/*

Veanes, M., d. Halleux, P. and Tillmann, N. (2010), Rex: Symbolic regular expression explorer, *in* '2010 Third International Conference on Software Testing, Verification and Validation', pp. 498–507.

Wilcox, J. (2018), 'File I/O in Dafny with fileio.dfy', `https://stackoverflow.com/a/52470751`. Accessed: 20109-04-01.

Project Specification

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.

---

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

> ⓘ 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 A.1: Projected work by time period. Deadlines for other modules included where known.

Table A.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 A.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 A.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 1990*, any static analysis of external code should only be conducted with permission. Discovered issues should be disclosed responsibly.