Software Engineering with Incomplete Information

David Kelly

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

Department of Computer Science
University College London

16th March 2022

I, David Kelly, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work.

Abstract

Information may be the common currency of the universe, the stuff of creation. As the physicist John Wheeler claimed, we get "it from bit" [213]. Measuring information, however, is a hard problem. Knowing the meaning of information is a hard problem. Directing the movement of information is a hard problem. This hardness comes when our information about information is incomplete. Yet we need to offer decision making guidance, to the computer or developer, when facing this incompleteness. This work addresses this insufficiency within the universe of software engineering.

This thesis addresses the first problem by demonstrating that obtaining the relative magnitude of information flow is computationally less expensive than an exact measurement. We propose *ranked information flow*, or RIF, where different flows are ordered according to their *FlowForward*, a new measure designed for ease of ordering. To demonstrate the utility of *FlowForward*, we introduce *information contour maps*: heatmapped callgraphs of information flow within software. These maps serve multiple engineering uses, such as security and refactoring.

By mixing a type system with RIF, we address the problem of meaning. Information security is a common concern in software engineering. We present OAST, the world's first gradual security type system that replaces dynamic monitoring with information theoretic risk assessment. OAST now contextualises

FlowForward within a formally verified framework: secure program components communicate over insecure channels ranked by how much information flows through them. This context helps the developer interpret the flows and enables security policy discovery, adaptation and refactoring.

Finally, we introduce SAFESTRINGS, a type-based system for controlling how the information embedded within a string moves through a program. This takes a structural approach, whereby a string subtype is a more precise, information limited, subset of string, *i.e.* a string that contains an email address, rather than anything else.

Impact Statement

Type systems form an integral part of many programming languages. The power, utility and importance of type systems is only underscored by the increasing prevalence of type checkers for dynamically typed languages. These type checkers for dynamic languages accept partially type annotated programs: given this incomplete information, they provide partial correctness guarantees. The advantage is they allow different degrees of precision during different parts of development process.

The thesis proposes the use of information theory and context free grammars to address the challenges of incomplete correctness guarantees for partially statically checked programs.

The first part of the thesis shows how to use information theory as a program analysis tool, unconstrained by types. *Ranked Information Flow* maps the risk associated with formally verified elements of a program communicating over potentially insecure channels. To show the generality of ranked information flow as a tool not just for security, it proposes *Information Contour Maps*, a novel program visualisation. It gives empirical evidence that information theoretic measures can be efficiently calculated via ranking during program testing. The thesis describes an open source prototype tool, RIFFLER, for producing information contour maps. The tool can already benefit software engineers, and further development will

further extend its power.

The next part of the thesis introduces a new optional type system. OAST (Optional Security Typing) is the world's first optional type system for security. OAST replaces the complex, and costly, dynamic semantics of gradual security type systems with information theoretic risk analysis. This approach solves the refactoring problem of security type systems, which require policy to be in place at all stages to ensure testing. Information theory replaces dynamic semantic alternations in the language and can be used during the testing phase, not during deployment. Security typed languages, while popular in research, have had no impact on software development practice. OAST should change this situation due to its great flexibility and its focus on Python, one of the world's most popular programming languages.

The final part of the thesis introduces a novel form of typing for strings, SAFESTRINGS. These permit a flexible subtype relation over strings. Strings are a common avenue for attack in software systems, as a string is unbounded in the information it can contain: SAFESTRINGS close this avenue. The use of subtypes constrains the entropy of a string, making programming with SAFESTRING a simpler task and constraining the possible error space. A prototype SAFESTRING library is available that can already improve the security of existing Java software. SAFESTRINGS have already had a real world impact, having been incorporated in the BOSQUE programming language from MICROSOFT. SAFESTRINGS have also been used to improve testing efficiency with EVOSUITE, and improved fuzzing of RESTful APIs with RESTLER.

Acknowledgements

First and foremost, to my wonderful wife and son, Tania and William. They have both tolerated a mixture of doubt, depression and despair; especially through the seemingly endless days of Covid. Thanks to EPSRC, for funding this entire thing through the DAASE project. To my supervisors, Earl and David: both provided many a thought-provoking conversation. Earl also managed to secure the funding for me and recognised whatever latent ability there might be.

Thanks also to Leo, for endless interesting conversations about computer science, music, politics, economics and life in general. Dan, for help with Java programming and SAFESTRINGS in particular. And special thanks to Rae, without whom none of this would ever have happened.

Contents

1 Introduction **1**

1.1 Challenges . . . 3

1.2 Basic Concepts . . . 5

1.3 Contributions and Outline . . . 9

2 Background and Literature Review **14**

2.1 Information Flow Control Background . . . 14

2.1.1 Noninterference . . . 15

2.1.2 Security Typing . . . 18

2.1.3 Flow Sensitive Type Systems . . . 24

2.2 Gradual Typing Background . . . 31

2.2.1 Cast Calculus . . . 37

2.2.2 Gradual Guarantee . . . 40

2.2.3 Abstracting Gradual Typing . . . 42

2.2.4 Extended Type Systems . . . 44

2.2.5 Gradual Typing Efficiency . . . 46

2.2.6 Static and Dynamic IFC . . . 48

2.2.7 Granularity . . . 51

2.2.8 Other Approaches to Information Security . . . 52

2.3 Gradual Information Flow . . . 54
2.3.1 Gradual Security Type Systems . . . 54
2.3.2 Quantified Information Flow . . . 59
2.3.3 Entropy Estimators . . . 60

3 Ranked Information Flow **62**
3.1 Introduction . . . 63
3.2 Motivating Example . . . 66
3.3 RIF: Ranked Information Flow . . . 69
3.3.1 Cost of Dynamic QIF . . . 69
3.3.2 *FlowForward* . . . 72
3.3.3 Interior Information . . . 74
3.3.4 Data and Entropy Estimators . . . 75
3.3.5 Noise Reduction via Clustering . . . 77
3.4 RIFFLER Implementation . . . 78
3.4.1 Capturing Observations . . . 79
3.4.2 RIFFLER Discussion . . . 80
3.5 Evaluation . . . 83
3.6 Case Studies . . . 88
3.7 Related Work . . . 93
3.7.1 Information Theory in SE . . . 94
3.7.2 Information Theory in Computer Security. . . . 95
3.8 Conclusion . . . 98
3.9 Appendix: Creating a corpus of Python Projects . . . 99

4 Optional Security Typing **100**
4.1 Introduction . . . 101

4.2 Background 105
4.3 Motivating Example 106
4.4 OAST Language 110
4.4.1 Core Language 110
4.4.2 Security Labels 112
4.5 Ranking Flows for Security 118
4.5.1 A Python Prototype 123
4.6 Noninterference and Confinement 128
4.6.1 Chinks in the Program 131
4.7 Related Work 139
4.8 Conclusion 143
4.9 Appendix: Proofs 145
4.9.1 Proof of Noninterference 145
4.9.2 Confinement Proof 148

5 SafeStrings **150**
5.1 Introduction 151
5.2 Motivating Example 155
5.3 SafeStrings 157
5.3.1 Definition 158
5.3.2 The SAFESTRING Subtype Relation 162
5.3.3 Equivalence of Operations over SAFESTRINGS and Strings . 164
5.3.4 Operation Overriding 166
5.3.5 Locally Gradual Typing 168
5.4 Realising SAFESTRINGS 172
5.4.1 Program Transformation 173
5.4.2 Constructing a SAFESTRING Library 174

5.4.3 Making Your Own SafeStrings 179
5.4.4 SafeStrings in Bosque 182
5.5 SAFESTRINGS: Evaluation 184
5.5.1 SAFESTRING Case Studies 184
5.5.2 The Cost of Adopting SAFESTRINGS 189
5.6 Related Work 195
5.7 Conclusion 201

6 Conclusion **202**
6.1 Contributions and Summary 202
6.2 Future Work 204

Bibliography **208**

List of Figures

1.1 Explicit flow 8

1.2 Implicit flow 8

2.1 Secure *while* language 19

2.2 Flow-sensitive type system 26

2.3 Flow-sensitive channel 29

2.4 Static semantics of GTLC 34

2.5 Type consistency for GTLC 35

2.6 Gradual function application 36

2.7 Function application in STLC 39

2.8 A failed cast with positive blame 40

2.9 Type and term precision for GTLC 41

2.10 Abstraction and concretisation in AGT 44

2.11 Syntax of λ_{gif}. 55

2.12 λ_{gif} operation semantics 55

3.1 ICM for a PYTHON malware 68

3.2 The effect of population on estimating mutual information 70

3.3 *FlowForward* Venn diagram 74

3.4 A clustering Galois Connection 78

3.5 Effort of ranking vs. exact 88
3.6 Accuracy of ranking over time 89
3.7 hmac ICM extract 90
3.8 ecsda ICM extract 91
3.9 idna ICM extract 93

4.1 Mutual information after 100 samples 110
4.2 Mutual information after 200 samples 111
4.3 λ_{OAST} grammar 112
4.4 Security label join 113
4.5 Security label meet 113
4.6 Label consistency 114
4.7 Label stamping 114
4.8 Type consistency rules for λ_{OAST}. 115
4.9 OAST inference rules; Figure 4.10 defines $\tilde{\nabla}$ and Figure 4.11 defines $<:$. S ranges over security types, *i.e.* base types labelled with either a concrete label or $\star$. 116
4.10 Joins and meets on types 117
4.11 λ_{OAST} subtyping rules. 117
4.12 Big step operational semantics for λ_{OAST}. 119
4.13 Four chinks in a PYTHON program 125
4.14 Confinement property inference rules. 138

5.1 String subtype hierarchy 163
5.2 String and SAFESTRING equality 166
5.3 Different stages of SAFESTRING annotation 170
5.4 Partial and complete SAFESTRING annotation 171

5.5 SAFESTRING program transformation 173
5.6 SAFESTRING library class diagram 176
5.7 Email SAFESTRING structure 178
5.8 Java code for parsing RGB strings 190
5.9 Type frequency in the 30 classes sampled from SF110. 193
5.10 The average *worst case* ϕ construction time for the SAFESTRINGS in our Java library. 194
5.11 Average *worst case* memory consumption for SAFESTRINGS. 194

List of Tables

2.1 Performance of gradual typing 47

3.1 RIFFLER results on the atheris fuzzer 87

Listings

1.1 Annotation removed from salary . . . 4
2.1 No sensitive upgrade . . . 27
2.2 Secure information flow rejected by Jif . . . 50
2.3 Assignment in a conditional . . . 53
2.4 LJGS syntax . . . 57
2.5 Dynamic LJGS . . . 58
4.1 An embedding of OAST into PYTHON . . . 123
4.2 Failure to confine . . . 141
4.3 A bad type check in TYPESCRIPT . . . 141
5.1 A string split on a colon . . . 156
5.2 Addition of a SAFESTRING annotation . . . 157
5.3 Code clarity: from 6 lines to 2 . . . 158
5.4 Carrying metadata in a SAFESTRING . . . 158
5.5 Concatenating SAFESTRING colours . . . 166
5.6 Slicing in TYPESCRIPT . . . 168
5.7 Well typed string code with errors . . . 170
5.8 Partial SAFESTRING annotation . . . 170
5.9 Complete SAFESTRING annotation . . . 170
5.10 Desugaring in TYPESCRIPT . . . 174

5.11 Dynamically generated error messages 175
5.12 A simple SAFESTRING 180
5.13 BOSQUE quick SAFESTRING creation 183
5.14 Invalid filepath 185
5.15 Preventing the passing of invalid strings 185
5.16 Subtyping for filenames 186
5.17 Mixing units of measure 188
5.18 Adding incompatible units 189
5.19 SAFESTRING code simplification 191
5.20 A BOSQUE calculator API 192
5.21 An object wrapped string. 196

Chapter 1

Introduction

Any software artefact that has either input or output exposes itself to security risks, just as a house with a door has a point weaker than other points. What comes out of the door may be a security concern, and what comes in the door may be an integrity concern. Just as a house without a door is difficult to use, software without I/O exists only in rare circumstances.

One approach to securing the information in a system is to measure the amount of information coming in and out of that system. This is akin to interrogating anyone entering or exiting our metaphorical house. This is a powerful method, as we can learn *exactly* what everyone knows. This gives the greatest flexibility to a software developer. Just like with interrogation, however, it is difficult and slow to conduct so thorough an investigation. This approach fails to scale when many people are coming in and out of the house, or communicating within the house. The solution is to spend more time with those people who look suspicious or whose behaviour is unusual, and less time with those who appear innocent. This technique is explored in Chapter 3.

Another approach to protecting the confidentiality and integrity of information is based on *security typed languages*, in which type systems are extended to

express security and integrity policies for programs and the data those programs manipulate. The compiler checks the policy before running the program and detects *potentially* insecure programs before they have a chance to do damage in deployment. Security-typed languages can enforce *information flow* polices, thus protecting data in a program. To continue the metaphor, while someone may come into the house, they cannot go wherever they like. Likewise, someone coming out of the house cannot come from just any room, nor even look at certain things or in certain rooms on the way out. Everybody in the house must wear a label that precisely dictates where they can go and what they can do. Not everyone wants to wear a label, and there might not even be enough labels for everybody. Addressing this problem motivates Chapter 4.

Developers want to track and limit the movement of certain forms of information within the program. This is what traditional, non security, type systems do: they prevent "cross contamination" of data by limiting the way certain types, say `string` and `number`, interact. Most type systems have an information black hole in the shape of the *string* type. An `integer` type can only hold an integer, but a string can contain *anything*: they are a universal encoding mechanism given "a free pass" by the compiler. To model the complexity of strings with a traditional IFC language would be clumsy at best. The security lattice (Chapter 2) required is liable to explosion. Usefully, many strings can be modelled, filtered, and labelled, with context free grammars. Just as with traditional IFC systems, these labelled strings can be mapped, tracked and traced. As their base type remains string, similar techniques for type system gradualisation can be applied to them (Section 5.3.5). If strings are an embedded calculus within a programming language, then that calculus alone can be gradualised inside a non-gradual type system. This is as an extremely flexible labelling mechanism, used instead of, or in addition to, any

existing IFC typing.

1.1 Challenges

Work on provably enforcing confidentiality and integrity policies in computer systems using security-typed languages is not wanting [141, 36, 182]. Many developers are familiar with at least the basic concepts of type systems. It is curious then, that one cannot find any security-typed languages used in industry. Researchers [13, 114] link this lack to usability problems. Some propose *gradual type systems* for information security (Chapter 2). Gradual security type systems allow partial disclosure of information policy. At runtime, such languages attempt to reconstruct an enforceable security policy. To conclude the house metaphor, some people in the house wear labels, whereas others do not. The problem is to solve the question of what the missing labels ought to be. Such an approach has a problem: the incompleteness of the typing information leads to inference problems.

To demonstrate this, consider the code in Listing 1.1 which might be suitable for a company's human resources software. It has partial security information: `salaryIncrease` is a public function, as indicated by the subscript l (low) on the function name. Assuming the existing labels are correct, `salaryIncrease` is a public channel that processes public variables. We calculate the return type of `salaryIncrease` to be Num_l. If we try to write the program `salaryIncrease( salary)`, we should have a static type error. Now, assume that g is some additional functionality that we wish to include. The program `salaryIncrease(g(salary))` now passes the static type checker. Of course, it is not 'correct': g has laundered the confidential information. A sound gradual system detects this laundering and behaves accordingly by producing a runtime type error.

```
/* calculate a salary increase */
salaryIncrease_l : Num_l → Num
salaryIncrease x = x + (mod x 5)
/* a public method still under development */
g : Num →_l Num
g(x) = x

salary : Num_h
salary = 35433
```

Listing 1.1: If we remove the annotation on `salary`, there is insufficient information to infer the intended security policy, making development and testing more difficult. Should *salary* be confidential or public?

A simple alteration makes resolving the policy in Listing 1.1 much more difficult. Neither optional nor gradual systems should require type annotations. Removing the annotation on `salary` should not introduce ambiguity. In this case, the removal *does* introduce ambiguity. Should `salaryIncrease(g(salary))` type check? Does it violate policy? The only way to solve this problem is to know the security label of `salary`. A gradual type system, enforcing a security policy at runtime, must make a decision about `salary`. Such a type checker can guarantee that a policy is enforced, but it cannot guarantee that the *developer's* policy is enforced: if the checker defaults to h, the program errors. If it defaults to l, then the runtime decides that `salary` becomes public. The utility of this approach to the software engineer is dubious; making strict guesses in the face of incomplete information risks false positives and is unlikely to be welcomed by the software engineering community. The problem is only likely to be exacerbated during the development stage, where frequent code changes and churn are likely to mean

the security policy is highly underspecified.

Intended policy cannot be reliably inferred in all cases. Even highly sophisticated IFC systems do not have complete type inference. JIF, an expressive security language built on Java [141] only supports inference for local variables and local parameters. Formal parameters for methods must always be annotated. In a partially annotated program, there is a family of inferable policies, only a subset of which model the intended policy. This thesis proposes an alternative method, where a partially annotated program is imagined instead as secure modules communicating via potentially insecure channels: these channels are assessed using the novel information theoretic technique of *ranked information flow*, also proposed in this thesis for the first time as a technique not just for security, but for general software engineering.

Listing 1.1 is a simple example where the security levels are just $l \sqsubseteq h$. Consider instead a program which tracks the integrity of public inputs. Such a program is likely to offer a public API. RESTful APIs are a good example. These are string dominated entry points to a program. Labelling *all* inputs via this API as of low integrity is too broad, as some may be harmless, whereas others may have carefully crafted attacks, such as arbitrary code execution or *Denial of Service* (DoS) attacks. Many strings may simply be ill-formed. Type systems are ill-equipped to deal with this situation, hence the large number of *ad hoc* solutions to string filtering and sanitation. This thesis shows how strings can be gradually typed in an existing language to flexibly track the integrity of strings within a program.

1.2 Basic Concepts

Some terms have already been used in the proceeding material, without definition. While the intuitive meaning of most of them may be easily grasped, this section

rectifies that lack and defines the most basic terms and concepts as used throughout this thesis. **Programs** consume **inputs** and produce **outputs**. In software testing, the standard term for a program that is undergoing testing is **Software Under Test (SUT)**. An input to a program may be a string, a numeric value, a file or any appropriate stream or object in memory. For generality, we consider programs that do not take an explicit input as taking a **null** input. We do not consider the case of a partially applied program. Every program and subprogram is assumed to have all of its inputs. In this sense, when the term program is used, it always refers to a closed λ-expression, or something which may be transformed into an equivalent closed λ-expression.

One possible definition for a **Type System** is "a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases to the kinds of values they compute" [153]. It is assumed that every **term** in a language has a type, either known via an **annotation**, or computable via some inference mechanism. Broadly speaking, there are two major branches to the study of type systems. These are the more practical, making type systems for programming languages, and the more abstract, which focuses on systems for various forms of typed **lambda calculi**. This thesis uses both approaches, with an emphasis on the practical.

A common distinction for type systems is between **static** and **dynamic**. A static type system relies on annotations and inference to determine the types of terms and check that there exists a typing derivation for the program with respect to a finite set of typing rules. A static type system does not have a dynamic checking element: indeed many languages erase typing information at runtime. A dynamic language performs type compatibility checks at runtime. The terms **strong** and **weak** are often applied to type systems. These terms

do not have agreed definitions. All type systems discussed in the thesis are informally described as strong: a term's type cannot be silently coerced statically or dynamically via the typing rules. Explicit casting, by the developer, at runtime is allowed.

A **Gradual Type System** allows for the types of terms to be only partially known statically. A gradual type system is a **hybrid** system, merging, to some degree, both static and dynamic typing. For those terms with statically unknown types, their types are resolved via dynamic methods during runtime. This thesis distinguishes between gradual type systems and **Optional Type Systems** by observing that an optional type system allows for partial static type checking, but does not introduce changes to the language semantics: an optional system is a one-sided gradual system. The terminological distinction between gradual and optional systems, while clear, is infrequently observed in actual systems, publications and discussions. Optional systems with no runtime component, such as those of TYPESCRIPT and PYTHON, are termed gradual. This is an unnecessary confusion.

An **Information Flow Control** (IFC) programming language is a language equipped with a type system intended to monitor the **security** and **integrity** of data. Data security means not allowing the influence of private data on a computation to be inferable from the public behaviour of the computation. Integrity is the dual of this: it controls the influence that possibly untrusted data can have on the behaviour of a program. Static type systems are the main approach to information flow control in programming language research. Gradual information flow control languages are a more recent edition to the research agenda.

Information flows come in two forms: *implicit* and *explicit*. An explicit flow is the result of assigning a secret value to a public variable. This is in violation

$$l := h;$$

Figure 1.1: The simplest explicit flow. The secret value is directly assigned to a publicly observable variable. In general, such flows are easier to detect than *implicit* flows.

$$\begin{aligned} &\text{if} \quad h = 1 \\ &\quad \text{then } l := 0 \\ &\quad \text{else } l := 1 \end{aligned}$$

Figure 1.2: An implicit flow through a conditional statement. The output is determined by the value of *h*, so its value can be inferred by looking at the result of evaluating the expression.

of the *no write down* principle of information flow [24]: data can only be written to memory at, or higher than, the security level of the data to be written. If we assume a policy that labels data as either $\top$ for secret or $\bot$ for public, with the partial ordering $\bot \sqsubseteq \top$ then writing from $\top$ to $\bot$ sends confidential information to a public location.

Implicit flows arise in control flow statements in a program (Figure 1.1 and Figure 1.2). It is worth noting that implicit flows *always* exist in control flow statements by necessity, but can be either *licit* or *illicit*, depending on policy. Implicit flows are preventable by disallowing a $\top$ guard in a conditional or while statement from effecting low data in the body of the conditional.

Flows move through information *channels*. These channels can take diverse

forms, such as *termination channels* and *power channels*. These flows are also implicit but are more rarely considered in the security type literature. This is doubtless due to the additional complexity involved in modelling the semantics in a type theoretic manner. If we consider a *side channel*, such as *resource exhaustion*, where information leaks via the exhaustion of a finite system resource, then this should be amenable to policing with a type system.

Information Theory was introduced by Claude Shannon [172] in 1948. Information theory provides a formal definition of information and a means to quantify it. The essential elements are the **Shannon Information**, or **surprisal**: a quantity derived from the probability of a particular event occurring in a random variable; and **entropy**: how much surprisal a random variable has on average. This thesis uses information theory to explore software and to address the shortcomings of optional type systems. One of the central techniques used in this endeavour is entropy estimation. Information theory is an extremely powerful framework for analysis, but has limitations due to the expense of providing high quality estimates. As the entropy calculation relies on knowing perfectly the distribution over a random variable, X, it may be clear that a perfect entropy calculation is almost impossible for any real world software. Knowing the input distribution for a program is rare, so much be approximated through testing.

1.3 Contributions and Outline

This thesis investigates the problem of incomplete information within software systems. The constraints on incompleteness increase as the thesis proceeds. Each chapter moves through an increasingly precise forms of analysis, starting with exploratory methods using information theory, and ending with a specialised embedded type system for strings. Chapter 2 introduces the lattice-model of

information-flow policies and all the notation used in this thesis. This chapter defines noninterference; the means by which a security-typed language protects the information inside a program. This chapter also defines both gradual and optional typing and how both are treated in the remainder of the thesis. This chapter also introduces sufficient information theory for the remainder of the thesis. This chapter is largely based on the existing work of other researchers and programming language technology designed to enforce information-flow policies and ensure that programs handle strings in a secure fashion.

In Chapter 3, the thesis addresses the problem of scaling information theoretic measures to handle full program analysis. This chapter shows new uses for information theory beyond merely security, including refactoring and testing adequacy. No existing research explores the problem of scaling information theory to software engineering, nor does it consider information theory from any perspective other than security. This chapter shows that there are multiple uses in software engineering for incomplete or uncertain information. It is the first work to explore the utility of ranking information theoretic measures rather than relying on producing tight bounds on information theoretic estimates.

Chapter 4 builds on the approach of Chapter 3 by incorporating ranked information flow into a type system for information security. This is entirely novel and without precedent in the literature. The cost of casting and asserting that one pays in a gradual type system transfers to testing instead. RIF provides the flexibility to refactor and rank risk in partially annotated programs, something which no existing type system is capable of doing. This system, OAST, is explored through a simple lambda calculus. Chapter 4 so that such an approach allows fully typed programs to have a formal proof of correctness, while assessing the risks in under-annotated programs.

Chapter 5 introduces a novel approach to more flexible and detailed typing for strings. It embeds a sound gradual typing system, with runtime checks, into the string type of an object-oriented language, in this case Java. Such an embedding is performed without any changes to the language, relying instead on a program transformation and support library. These can be used solely during the testing phase if so desired. By not gradualising every type, with the potential of any type to be any other type, SAFESTRINGS avoid much of the complexity of the dynamic component of gradually typed programming languages. SAFESTRINGS provide fluid subtyping for strings than allows the user to go up, or down in typing precision. SAFESTRINGS have already been incorporated as a native element into the BOSQUE programming language from MICROSOFT.

The main contributions of this thesis are:

- The first investigation in ranked information flow; a means to reduce the cost associated with sampling-based estimation of information theoretic measures. It introduces a new information theoretic metric, *FlowForward*, which is more ranked more meaningfully than Shannon measures.

- The introduction of *Information Contour Maps* (ICMs), heat-mapped call-graphs which visualise the movement of information through the SUT. A tool, RIFFLER, is introduced, that provides the necessary information to build an ICM from a Python program. Various uses for an ICM are detailed, along with a means to build them via fuzzing, thus requiring no extra testing resources. ICMs use a new information theoretic measure, *FlowForward*, interpreted as an asymmetrical form of mutual information. This measure is easier to rank meaningfully than existing information theoretic measures.

- The introduction of optional information security, OAST. OAST is the first optional type system for IFC. It incorporates testing into its decision procedure by using RIF, to rank areas most at risk of high information flow. By replacing the strict requirements of dynamic gradual typing, OAST solves the refactoring dilemma. OAST builds on the work of Chapter 3 by providing a new stopping condition for the RIF algorithm. This new stopping condition harnesses the fact that the Bayesian estimator used by RIFFLER returns a distribution. In any system which has more than one observation point (and therefore can be ranked), then the standard deviation of the distribution, and the Wasserstein distance between the distributions can be used to provide an arbitrary degree of confidence in the quality of the ranking.
- SAFESTRINGS are a new means to increase type safety for strings. They provide a novel means to build a subtype relation under the `string` type. This is achieved without creating any new typing extensions to a programming language. They can be embedded as a testability transformation into any existing OOP language. We evaluate SAFESTRINGS through case study, annotation burden, and efficiency. We introduce a programming language from MICROSOFT, BOSQUE [128], which has built-in support for SAFESTRINGS.

Finally, Chapter 6 concludes with a summary of the contributions and some future directions. This thesis introduces ranked information flow to software engineering, as a means to explore, not just security policies, but also to suggest changes and improvements to the software and its environment. It also introduces the theory of improving guarantees for optional type systems, specifically with respect to information security and string-manipulating programs. In addition to

the work in this thesis, the author has two papers under preparation, as co-author, on the effects of SAFESTRINGS on testing efficiency.

Chapter 2

Background and Literature Review

This chapter introduces the lattice model for specifying levels of confidentiality and integrity for data in programs. It introduces the definitions and basic theory of gradual typing, with special reference to gradual typing for secure information flow languages. As this thesis takes a hybrid approach to gradualisation, this chapter also introduces the required elements of information theory. Finally, we introduce the concept of locally gradual typing, and how it can be applied to string manipulating programs.

2.1 Information Flow Control Background

Security typed languages provide a means to add annotations to language terms. These annotations specify the confidentiality and integrity of the term. For example, the declaration `secret : int h` indicates that `secret` is an integer value with confidentiality level `h`.

Following work on multilevel security [25] and the work of Denning [68, 67] on program analysis, the security levels available via annotation should form a

lattice.

Definition 2.1.1 (Lattice)**.** A **lattice** $\mathscr{L}$ is a pair $\langle L, \sqsubseteq \rangle$ where L is a set of **elements** and $\sqsubseteq$ is a reflexive, transitive and anti-symmetric binary relation on L. In addition, for any subset X of L, there must exist both a least upper and greatest lower bound *w.r.t* the partial order $\sqsubseteq$.

An **upper bound** for a subset X of L is an element $\ell \in L$ such that, for all $x \in L$, $x \sqsubseteq \ell$. The **least upper bound** or **join** in X is an upper bound ℓ such that, for any other upper bound $z \in X$, $\ell \sqsubseteq z$. The notation $x_1 \sqcup x_2$ is used to denote the join of two elements x_1 and x_2.

A **lower bound** for a subset X of L is an element $\ell \in L$ such that, for all $x \in L$, $\ell \sqsubseteq x$. The **greatest lower bound** or **meet** in X is a lower bound ℓ such that, for any other lower bound $z \in X$, $z \sqsubseteq \ell$. The notation $x_1 \sqcap x_2$ is used to denote the meet of two elements x_1 and x_2.

As a lattice is required to have a join for all subsets of L, there must be a join for L itself, which is the **greatest** element; this is denoted $\top$ and called the **top** element. Likewise, there must exist a **least**, or **bottom** element, denoted $\bot$.

2.1.1 Noninterference

Noninterference, NI, has its origins in the work of Goguen and Meseguer [95], which itself builds on the secure information flow models of Denning [68]. The intuition for NI is relatively simple, but it has taken on a vast number of forms which results in many difficult choices for anyone attempting a practical implementation of IFC [167, 94, 123, 30, 198, 59]. Much IFC research concerned with type-system enforcement has focused on termination-insensitive noninterference (*Termination Insensitive Noninterference*). *Termination Insensitive Noninterference* guarantees NI up to correct program termination. If a program terminates in an error condition,

or fails to terminate at all, then *Termination Insensitive Noninterference* makes no claims about leakage of sensitive information.

To formalise this intuitive presentation of *Termination Insensitive Noninterference* it is necessary to assume a simple *security lattice*, where high, H, is secret data and low, L, is publicly readable. The only disallowed flow in this simple lattice is from H to L. Then *Termination Insensitive Noninterference* for a deterministic program is formulated by Volpano *et al.* [209] as:

Definition 2.1.1 (Noninterference)**.** Program c satisfies noninterference if, for any memories μ and ν that agree on low variables, the memories produced by running c on μ and on ν also agree on low variables (provided that both runs terminate successfully).

Note that this definition only compares memories before and after a *successful* execution run. Other details of execution are ignored. We might be able to learn something about program c by measuring how long a computation takes (thus learning something about μ and ν) or via some other *side channel*. *Termination Insensitive Noninterference* does not account for this possibility. Termination-*sensitive* noninterference, on the other hand, guarantees that even if the program fails to terminate properly no sensitive information leaks. *Termination Insensitive Noninterference* is weaker than this, but it is therefore easier to write a program that satisfies its conditions. One might consider the price of *Termination Insensitive Noninterference* with respect to termination-sensitive NI to be too high: exactly how much information can leak under *Termination Insensitive Noninterference* in the event of divergence has been studied by Askarov *et al.* [12]. They show that *Termination Insensitive Noninterference*, traditionally thought of, without justification, as leaking only one bit of information (in the information theoretic sense of bit), can in fact leak all of its secrets. This is not a terrible as it seems: they also

prove that an attacker would need to mount a brute-force attack in order to gather any sizeable body of information. This ultimately makes this potential channel a largely impractical form of attack, as the attacker cannot reliably learn a secret in time polynomial in the length of that secret.

Sabelfeld and Myers, in their excellent survey [165], give a more general formalism of noninterference, not *Termination Insensitive Noninterference* specific:

Definition 2.1.2 (NI: Sabelfeld and Myers)**.** A program C can be considered noninterferent iff

$$\forall s_1, s_2 \in S. s_1 =_L s_2 \Rightarrow [\![C]\!]s_1 \approx_L [\![C]\!]s_2$$

Two possible initial states, s_1 and s_2 which have equivalent *low* initial values, are noninterferent if their behaviours, as defined by the language semantics, are indistinguishable to an attacker. The *low-view* relation $\approx_L$ offers the flexibility to define what behaviours are being compared for distinguishability and says that the two states, S_1 and $_2$ are indistinguishable to an observer up to the limits set by the relation.

Statically enforced type systems, such as that proposed by Volpano *et al.* [209] (Section 2.1.2) can enforce *Termination Insensitive Noninterference*, but are so restrictive that they have seen negligible real world use [13]. Work has been done to alleviate some of these restrictions [185] but again, they have seen little practical use in any mainstream language. The relative lack of uptake in industry, despite the prevalence of security leaks and bugs, strongly argues for a new approach to the implementation mechanisms of IFC, rather than a new approach to noninterference.

2.1.2 Security Typing

Now that we have sufficient understanding of the problem we are trying to solve, we move on to various solutions mediated through the prism of type systems: either static, dynamic and hybrid.

Secure type systems are an instance of information flow analysis. Data are given security labels, which dictate their legal interactions with other labelled data. Security labels should form a *complete* lattice $\mathscr{L}$. We shall call such a complete lattice a *security lattice*. If ℓ and ℓ' are points in the lattice, then $\ell \sqsubseteq \ell'$ indicates that the label ℓ' is at least as restrictive as the security label ℓ. Data dependent on sources with labels ℓ and ℓ' must have a security label at least as restrictive as the join of ℓ and ℓ', $\ell \sqcup \ell'$. The literature, for the sake of simplicity, usually presents only two security labels, H and L, for high (secret) and low (public) respectively. In such a lattice the only disallowed flow is from H to L. This can be generalised to any complete lattice without loss of generality.

Figure 2.1 details a type system as presented by Sabelfeld and Myers [165] that is equivalent to Volpano *et al.* [209]. Security labels form the simple $L \sqsubseteq H$ security lattice. The label *pc* in Figure 2.1 refers to the *program counter*. This is a standard technique for controlling implicit flows. It indicates the security context in which the computation occurs. Programming in this framework is unforgiving: it is not intended to be used as a real-world language. It is, rather, a translation of Denning's certification conditions [68] into type inference rules. Volpano *et al.* use the proof of soundness of their system as a novel validation of Denning's lattice model.

The simple type system in Figure 2.1 does not protect against termination channels. A program with a high guard in a conditional expression (or in a while loop) can reveal information that guard in the event that the program fails to ter-

$$\vdash exp : high \qquad \frac{h \notin Vars(exp)}{\vdash exp} \qquad \text{[E1–2]}$$

$$[pc] \vdash \text{skip} \qquad [pc] \vdash h := exp \qquad \frac{\vdash \text{exp} : low}{[low] \vdash l := exp} \qquad \text{[C1–3]}$$

$$\frac{[pc] \vdash C_1 \qquad [pc] \vdash C_2}{[pc] \vdash C_1; C_2} \qquad \frac{\vdash \text{exp} : pc \qquad [pc] \vdash C}{[pc] \vdash \text{while } exp \text{ do } C} \qquad \text{[C4–5]}$$

$$\frac{\vdash exp : pc \qquad [pc] \vdash C_1 \qquad [pc] \vdash C_2}{[pc] \vdash \text{if } exp \text{ then } C_1 \text{ else } C_2} \qquad \frac{[high] \vdash C}{[low] \vdash C} \qquad \text{[C6–7]}$$

Figure 2.1: A simple security type system for a *while* language. The *pc* is the *program counter*, the security context in which a sub-expression is evaluated. There are two types of rule, *expression* rules and *command* rules. Some commands are typable in any context (C1 and C2), while others are context dependent. Assignment, $l := exp$, only succeeds when *exp* is *low*.

minate successfully. It guarantees *Termination Insensitive Noninterference*. Although Volpano and Smith [210] address this shortcoming by forbidding *high* guards in loops and conditionals, most subsequent work in type-based security has been variations on a theme of *Termination Insensitive Noninterference*. Smith [186] gives an excellent overview of how a type system can ensure noninterference properties.

This simple type system controls information flow, but does not allow for *flow sensitivity* or the declassification: the *down-casting* of private data (Section 2.1.3). The inability to declassify or reduce the classification level of information makes it

impossible to write something as simple, and necessary, as a password-checking program. A failed login attempt reveals some information about the password, though admittedly it is likely to be very small. One would expect a password application to be one of the natural homes of IFC applications, but such a program necessarily leaks information.

Flow-sensitivity is related to, but separate from the question of declassification. Flow-sensitivity allows locally insecure computations in a context where the *overall* flow is secure. Declassification is the deliberate (partial) release of confidential information. In the Volpano system, a security label for a variable remained as initially declared throughout the execution of the program. Recognition of this problem has led to the development of flow-based security labels [108], where the value of a label can change given the control flow of the program's execution. See Section 2.1.3 for a fuller discussion. In Volpano's system data can only become more classified during execution. This problem has attracted the name *label-creep*.

The important area of declassification has been much researched, and there is a good survey (up to 2009) of the field by Sabelfeld and Sands [167]. They discuss declassification along four different axes: *what* information is releasable, *who* has the authority to release the information, *where* the information is released, and *when* it is released. Declassification is usually implemented in a type system by the inclusion of a declassification primitive in the language. This has the power to coerce labels in either direction. The problem with such a language feature then becomes how to police access to the primitive. From the point of view of logic, such a primitive allows us to prove any flow satisfies noninterference: it injects inconsistency into the logic which can then be exploded.

Controlling declassification has lead to various suggestions: *relaxed noninterference* of Li and Zdancewic [123], robust declassification [224] and, perhaps

most importantly, to the *decentralised label model* (DLM) of Myers and Liskov [140]. DLM is especially interesting in that it forms the basis for the Jif compiler [141]. Jif is an information flow control dialect of Java. It provides extremely fine-grained control of information flow without the need to explicitly create a security lattice. The lattice arises naturally from the interaction of labels, owners and readers. In this model a principal claims ownership of an element in a program. In effect it allows a programmer to label an element of the system with an owner (or owners) which can dictate how the element's data may be declassified, and by what. This is extremely expressive, as policies may be delegated to other principals in an *acts-for* relationship. A label in Jif is in fact a construct of a *reader policy* and a *writer policy*. The reader restricts how information may flow from a principal, and the writer restricts how information from other principals is treated. This expressiveness has its downside in the form of substantial complexity (*cf.* Section 3.1).

Several mainstream languages already have information-flow analysis dialects, most notably Jif [141] for Java, as we have seen, and FlowCaml [182, 183] for OCaml. Neither of these dialects however can be described as being mainstream in the sense of being actively used by the programming community at large. Case studies of programming with Jif [106, 13] highlight certain problems, and suggest solutions, for programming with IFC types. As far as we are aware, Askarov and Sabelfeld's work on an online poker game [13] still remains the largest and most detailed case study of IFC programming in Jif. They found that while Jif was able to identify many potential security problems with security-critical code, it also required time to master and decreased developer productivity. Given more experience and appropriate IFC design patterns, productivity might move back in the direction of the baseline development time. This hypothesis is untested however. Addressing the different forms of declassification proved to be the

most challenging area of Jif programming, as its treatment of declassification was insufficiently flexible to account for the different axes of declassification. It is hard to imagine what a gradual version of the DLM might look like. Even though Jif has polymorphic labelling, which helps somewhat to reduce the annotation tax, it does not reduce the stringent requirements of noninterference. A type checker with added functionality for *sound* gradual security would be a complicated enterprise, and it is likely that type checking in the presence of only partial information might be an undecidable problem.

Various experimental languages have been developed, perhaps most notably KLAIM [146], designed for use in distributed systems and Paragon [34], another Java dialect. Spark, a dialect of Ada, uses a form of information flow analysis [42]. The SIF framework of Chong *et al.* [48] is another approach to a concrete application of language based information-flow control. This aims at building secure web applications, rather than as a generalised framework for secure programming. It is built on top of Jif. As already mentioned, Jif uses the *decentralised label model* of Myers and Liskov [140]. Thus it can naturally be used to model mutual *mistrust* between processes, vital in web applications. Reader policies describe confidentiality, and writer policies integrity. Integrity can be seen as the dual of confidentiality, it is public input which carries the possibility of taint and therefore has more restrictions placed on it. In this sense, low has become high.

Jif and SIF prevent the accidental or unintentional downgrading of information, and any information that is downgraded needs the permission of all principals involved. Chong *et al.* also develop Swift [47] for secure web applications. This too is built using Jif. Swift automatically partitions code between JavaScript running in the browser, and secure Jif code on the server. Swift seeks to make web applications which are 'secure by construction'.

Jif does not answer every question about policy management however. Swamy *et al.* [192] present *Rx*, which permits security policies to be actively updated during program execution. They achieve this by defining labels as *roles*. A role is a set of principals, and its ordering in the security lattice is defined by subset inclusion. Rx permits roles to be updated dynamically. To prevent covert channels forming through the change of roles, Rx also uses *metapolicies* which define which principals can view a role, and which principals trust a role's definition.

King *et al.* [114] have conducted work into the disconnect between practical applications of IFC, such as taint checking in Perl and Ruby, and the more purely theoretical work, focused on noninterference, which dominates the research literature. Focusing on Java (specifically Jif), they ask the question whether the extra cognitive and programming effort to identify and control *implicit* flows is efficacious. The conclusion they reach is that, with present systems, the number of false positives (detection of implicit flow where there are none) is uncomfortably high, and makes programming more difficult than at present. In the case of Jif, the majority of false alarms, where the compiler incorrectly identifying a leak where none exists, was due to the inability of Jif to prove the safety of null pointer access. It is perhaps worth noting that the majority of criticisms that King *et al.* level are aimed at languages which are predominantly imperative; they do not directly address the case of purely functional languages, or those that are immutable by default. They do suggest though that the absence of the null pointer from ML might be a beneficial feature that could be applied to the underlying semantics of the Java language. Better annotations can reduce the number of false positives from the Jif compiler, but providing them all initially is a large cognitive effort. Gradual typing, as we have seen in Section 2.2, given its focus on progressive

and iterative refinement of type information, provides a solution to this upfront cognitive cost.

2.1.3 Flow Sensitive Type Systems

We have seen that Volpano-style typing is inflexible in its policy enforcement and briefly described some real-world language implementations which seek to find principled means to overcome these limitations. We shall now look at some of these limitations and solutions in more detail, with a particular regard for increased expressiveness and ease of use.

It would be nice to devolve IFC to some runtime checking system, thereby absolving the programmer of the necessity for typing. Information flow is not, however, a dynamic property: it is not usually possible to detect an implicit flow simply by observing the behaviour of a running system. A dynamic monitor can only 'see' the result of one execution run, whereas NI is a *hyperproperty* [59]: it holds not over a single program trace, but a set of program traces. A program has to be executed multiple times to have sound dynamic NI.

Monitoring IFC purely dynamically is hard, but then so is handling *changes to policy*. Without some form of dynamic management, it would not be possible to implement policy changes without taking a system entirely offline and rewriting it to reflect the new policy. This is hardly a practical solution. Dynamic changes to the security policy demand some form of dynamic management. Organisational hierarchies are not static, so any IFC system, especially one with little or no downtime, must be able to account for variations in access policy and whether the data is trusted or untrusted at any given moment. In effect security labels must be flexible, changing depending on context. We call such flexibility *flow sensitivity* and there are a number of subtleties which must be handled to avoid introducing new covert channels.

The impact of gradual typing on flow sensitivity has not been adequately studied (Section 2.3). Flow sensitivity and graduality together would add two levels of ambiguity which need to be handled in a principled manner by a type checker. It is highly likely that any approach which required end-to-end soundness at runtime would have recourse to a dynamic monitor. This would necessarily have implications for runtime efficiency (Section 2.2.5).

Hunt and Sands [108] give a good overview of flow-sensitive security types. An important observation they make is that, at least as of 2006, most type systems for IFC were *flow-insensitive*; *i.e.* the order in which events occur is ignored by the IFC analysis. They give a nice example for intuition: an analysis is flow-insensitive if the results of analysing C_1; C_2 is the same as that for C_2; C_1. For a program to be secure within a flow-insensitive analysis *every* subprogram must also be secure. The program `secret; secret=0; l:= secret;` is secure in terms of its information flow as the final assignment of l does not reveal anything about the *initial* value of *secret*, but would be rejected by Volpano as the high security label associated with secret cannot be altered, even though it has been overwritten with the public constant 0. Type systems similar to that of Figure 2.1 would reject this as not well-typed. It is hard to imagine a useful application for flow insensitive IFC in industrial use. In practice, we ought to consider only type systems which allow for flow sensitivity and by extension declassification.

Figure 2.2 details a simple flow-sensitive type system for a *while* language, as given by Russo and Sabelfeld [162, p. 4]. The key detail to take from the type system is that of assignment. Assignment always type checks (*cf.* the Volpano system, where this is not the case). However the security level of the variable now changes to the *join* of the security level of the expression and that of the program context *pc*.

$$pc \vdash \Gamma \ \{\text{skip}\} \ \Gamma \qquad \frac{\Gamma \vdash e : t}{pc \vdash \Gamma\{x := e\} \quad \Gamma[x \to pc \sqcup t]}$$

$$\frac{pc \vdash \Gamma\{C_1\}\Gamma' \qquad pc \vdash \Gamma'\{C_2\}\Gamma''}{pc \vdash \Gamma\{C_1; C_2\}\Gamma''}$$

$$\frac{\Gamma \vdash e : t \qquad pc \sqcup t \vdash \Gamma\{c_i\}\Gamma' \qquad i = 1,2}{pc \vdash \Gamma\{\text{if } e \text{ then } c_1 \text{ else } c_2\}\Gamma'}$$

$$\frac{\Gamma \vdash e : t \qquad pc \sqcup t \vdash \Gamma\{e\}}{pc \vdash \Gamma\{\text{while } e \text{ do } c\}\Gamma} \qquad \frac{\Gamma \vdash e : t \qquad pc \sqcup t \sqsubseteq \ell}{pc \vdash \Gamma\{\text{output}_\ell(e)\}\Gamma}$$

Figure 2.2: A flow-sensitive type system similar to that in Hunt and Sands [108] but extended with and *output* operation. The most salient difference from Figure 2.1 is the handling of *assignment* (Rule 2). Assignment always type checks, but the security level of the variable is now the *join* of the *program counter, pc*, and the level of t.

Hunt and Sands also describe a family of flow-sensitive type systems for a simple *while* language [108]. They allow relabelling data to the join of the original label and a new label. This has to effect of increasing the classification of a datum. To move in the opposite direction, they introduce fresh variables into the context with a copy of the secret value now stored at a public level. This note the similarity of their method to *Static Single Assignment*. They show that their system is equivalent to Amtoft and Banergee's Hoare-style independence logic [9]. They further demonstrate that for any program typeable in a flow-sensitive system,

```
var x = false;
if x_h then y = true;
return true;
```

Listing 2.1: No sensitive upgrade is a simple method for dynamic IFC enforcement, but it rejects this secure program.

it is possible to construct an equivalent program which is typeable in a simple flow-insensitive system through the expedient of introducing extra variables. They present a code transformation algorithm that can automatically move a flow-sensitively typed program to an equivalent flow-insensitive program. Such a transformation must happen at compile time [162]. This might have utility in a gradual setting, as reducing the number of cast insertions into the code would have a beneficial outcome on performance. However, they also present a *worst case scenario* for the translation which would, if utilised naively, result in inefficient code. Hunt and Sands do not discuss how to check that a declassification is valid *w.r.t* policy, but apply their methods on the assumption that such relabelling in acceptable.

Many methods have been proposed to allow for purely dynamic floating-label IFC enforcement. Austin and Flanagan [15, 17, 19] have done a lot of work looking at purely dynamic information flow analysis. One of their key observations is that security labels, especially in a browser setting, exhibit *label locality*, where the majority of labels in a data structure have the same security level. This allows them to propose a *sparse labelling* semantics which leaves labels implicit whenever possible. There is a potential overlap here with gradual security typing where labels can be left implicit rather than explicitly annotated. This *coarse granularity* (Section 2.2.7) of IFC in certain settings has a relationship

with the modular gradual typing of Typed Racket. Exploiting sparse labelling semantics can greatly reduce the number of cast insertions into code, by treating modules as having one security label. Computation *within* a module could then be treated as essentially label-less (everything having to same label, or at least in a permissive flow-sensitive setting) and *inter*-module computation could then be handled, at least in the simple case, with a flow-insensitive type system. Fennell and Thiemann indeed propose something along these lines for gradual Java security [78, 79], as detailed in Section 2.3.

Zdancewic [223] considers a method called *no-sensitive-upgrade*, or NSU (Listing 2.1). Implicit flows are handled by prohibiting the leakage of data in a termination insensitive manner: in effect the runtime monitor halts execution of a program and throws an exception in the event that a high variable has an effect on a low variable. As it acts in instances where an information leak *might* happen, without reference to context, this method can be overstrict and reject too many secure programs. To address this problem, Austin and Flanagan [17] propose a *permissive-upgrade* strategy. In effect, this permits partial information leakage, but labelled appropriately and subsequently tracked throughout the program execution. Partially leaked data, labelled P, needs to be reclassified with a *privatisation operation* before reusing it in a conditional test. The effect is that it accepts more programs as secure and guarantee termination-insensitive noninterference, and this in a purely dynamic manner.

The use of dynamic labels can open up a new covert attack channel, the *flow-sensitive* attack. Buiras *et al.* [37] provide a simple example of the dangers of *arbitrary* label change, even if that change is to a more restrictive point on the security lattice. Consider the code sample in Figure 2.3, where variables are either l or h for low or high respectively, as marked by subscripts.

$$
\begin{aligned}
&l_l && := \mathit{True} \\
&tmp_l && := \mathit{False} \\
&\text{if } h \text{ then } (tmp_h && := \mathit{True}) \text{ else } \mathit{skip} \\
&\text{if } \neg tmp \text{ then } (l := \mathit{False}) \text{ else } \mathit{skip}
\end{aligned}
$$

Figure 2.3: A flow-sensitive channel. Inspection of the value of the public variable l at the end of the computation reveals the value of the secret boolean h.

If the secret value contained in h is *True* then *tmp* is promoted to a high variable and has its value set as *True*. If however the value of h is *False* then *tmp* is not reassigned and relabelled, and l takes on the value of *False*. The security label of l does not change in either case, but in the latter the value of h has now been leaked into the public variable l.

LeGuernic *et al.* [100] propose an automata based model for dynamically monitoring the flow of information during the execution of a program. This uses the results of a static analysis to accept or reject, at runtime, the execution of a program. It achieves this by sending abstractions of the program state to the automaton, which uses these abstractions to analysis the information flow within the system. The monitoring mechanism alters the program in the event that it infers an infringement of noninterference, thereby ensuring a *Termination Insensitive Noninterference* execution. This is reminiscent of the AGT approach to gradual typing (Section 2.2.3), where the results of a static analysis are used to provide initial proofs of plausibility, which are then dynamically elaborated at runtime in the form of a variation of unification.

Russo and Sabelfeld [164] suggest a hybrid static-dynamic ground approach by using a security type system and a monitor to reduce the number of false positives. Askarov and Sabelfeld [14] develop a hybrid policy framework which supports both termination-sensitive and insensitive noninterference policies. Other approaches to flow-sensitivity include that of Broberg and Sands [35], who propose the use of *flow locks*, a system where users of a high security level always have access to a given data, but users from a lower level can only access it when that data is *unlocked*. This also goes some way to addressing the complexity issues associated with the various types of declassification, especially that of the decentralised label model. Building on their initial idea, Broberg *et al.* [36] elaborate the concept of flow locks and propose *paralocks*. Paralocks is a language for creating fine-grained, static IFC policies. It is a very powerful framework, subsuming the earlier work on flow locks, but also allowing for the encoding of complex IFC systems, such as the decentralised label model. Paralocks have actually been used as the basis of a real language implementation, *Paragon* by Broberg *et al.* [34]. Paragon is a Java-based language which supports static information flow control as a first class concept. One of the chief advantages of offering another Java extension is that it allows a direct comparison with Jif. The main advantage of Paragon over Jif, according to its creators, is the increased flexibility of the system. Jif has the DLM 'built-in', allowing a single declassification construct, whereas Paragon can provide different IFC methods for different needs: that which is *hard-wired* in Jif is just a special case in Paragon. The authors say that they have encoded the entirety of the DLM in the form of a Paragon library.

2.2 Gradual Typing Background

Types are a lightweight tool used to verify certain aspects of a program. Languages often adopt either a static or dynamic type system to police undesirable behaviours. Generally speaking, static type systems allow for early detection of errors and can often result in a program which can be compiled to more efficient machine code. However they can have a slow development cycle and the inflexibility of a type system's logic can sometimes lead to the inability to write a program in a way that seems most natural to the programmer. Dynamic languages, on the other hand, foster a faster development cycle and a more flexible approach to experimentation, with the cost that errors might not be revealed until runtime. The nature of the error can often be opaque, and its origins specifically as a *type* error obscured.

While gradual typing has been characterised as the discipline of taking a static type system and conservatively extending it with an unknown, dynamic resolved type, it has its intellectual origins in adding types to dynamic languages, specifically Smalltalk and LISP. Strongtalk, by Bracha and Griswold [32], is a notable early entry in the family tree. Bracha has subsequently developed this idea into *pluggable* types [31]. The aim of gradual typing combines these two goals: it combines the advantages of static and dynamic typing disciplines in a principled fashion within a single language. Even though 'gradual typing' as a term has only emerged since the mid-2000s, the notion of embedding dynamic types within a static type system is far older. Abadi *et al.* [1] suggested the embedding of dynamic typing within statically typed languages as early as 1989. This suggestion was soon followed by Thatte's 'quasi-static' typing [199], which introduced a dynamic type mapped to the top ($\top$) element in a type lattice and

provided an accompanying algorithm to check that casts from $\top$ to a point lower in the lattice were 'plausible'.

Gradual typing did not emerge as a unified field and has failed to become one. Its initial efforts as a distinct research topic came with the near-simultaneous publication of two independent papers in 2006. The first, by Siek and Taha [176], was inspired by problems they had found in the 'quasi-static' typing methodology. Specifically, they describe gradual typing for a functional language with structural types. The result is a simply-typed λ-calculus with a type system sound with respect to fully-annotated terms. The second approach, by Tobin-Hochtstadt and Felleisen [202] adds types at the module level and uses constraint solving to decide when casts need to be inserted into unannotated code. The two methods work at different levels of granularity.

Gradual typing, on both tracks, is making partial industrial inroads. Various languages and language dialects already exist which incorporate some notion of optional typing, such as *Bigloo*, a Scheme dialect from INRIA, but the work of Siek and his students especially has been influential in the formalisation of gradual typing and in defining the properties a gradually typed language should have. Type systems going under the name 'gradual' have recently begun to enter the mainstream, most notably with Microsoft's TypeScript and both Flow [76] and Hack [77] from Facebook. They are not gradual in the Siek sense, however. To distinguish this we shall refer to systems such as Flow and Hack as having *optional* typing. Gradually typed languages in the pure sense exist in research, such as [29, 208, 99, 203].

Gradual typing, at least in Siek's view [179], is an approach to type safety characterised by *progressive refinement*. It achieves this by introducing a new

primitive type, *dynamic* or $\star$ [1]. The dynamic type is statically interpreted as a type 'wildcard': one which is unknown at compile time. We shall call non-$\star$ types *concrete types*. At runtime the type of $\star$ is concretised via casting or other means. Accompanying the dynamic type is the important notion of *type consistency*. This tests the plausibility that $\star$ is of an appropriate concrete type. Unannotated variables are usually assigned the value of $\star$. Figure 2.4 shows the inference rules for a gradual simply typed λ-calculus. Siek and Taha's system aims for full and smooth interaction between typed and untyped code.

Gradual typing is about plausible reasoning in the face of partial or imprecise information, using what is available at compile time to statically reject or optimistically accept a program, and performing additional checks at runtime to concretise the unknown elements. For example, a gradual function type $f : Int \rightarrow \star$ is more informative than $f' : \star \rightarrow \star$ and provides enough information to reject statically many ill-typed programs, such as $f + 1$. Indeed $f' + 1$ can also be rejected statically, as the 'shape' of f' is sufficient for rejection. It still accepts valid function calls, such as $f 1$ [122]. $f' 1$ will be accepted statically, as will $f' \text{``}one''$. We shall call this *optimistic* typing. An inappropriate return value should result in a runtime type error[2].

The simple gradually typed lambda calculus as originally presented by Siek *et al.* [176] as Figure 2.4, where $\lfloor \tau \rfloor$ represents either the concrete type τ or $\perp$, and $\sim$ is *type consistency*. Figure 2.5 shows the rules for type consistency as originally presented. Intuitively, types are consistent when they agree on those parts which are fully defined, or where one type can plausibly be the same or a subtype of the other. The dynamic type is (statically) consistent with anything. Type

[1] Also presented as ? in the literature. We shall use $\star$, pronounced 'star' or 'any' throughout.

[2] Optionally typed languages, such as TypeScript, do not attempt to provide to provide runtime type errors in this fashion, relying instead on JavaScript's native error mechanisms.

$$\boxed{\Gamma \vdash_G e : \tau}$$

$$\frac{\Gamma x = \lfloor \tau \rfloor}{\Gamma \vdash_G x : \tau} \quad (\text{GVar})$$

$$\frac{\Delta c = \tau}{\Gamma \vdash_G c : \tau} \quad (\text{GConst})$$

$$\frac{\Gamma(x \mapsto \sigma) \vdash_G e : \tau}{\Gamma \vdash_G \lambda x : \sigma.\ e : \sigma \mapsto \tau} \quad (\text{GLam})$$

$$\frac{\Gamma \vdash_G e_1 : \star \qquad \Gamma \vdash_G e_2 : \tau_2}{\Gamma \vdash_G e_1\, e_2 : \star} \quad (\text{GApp1})$$

$$\frac{\Gamma \vdash_G e_1 \tau \to \tau' \qquad \Gamma \vdash_G e_2 : \tau_2 \qquad \tau \sim \tau}{\Gamma \vdash_G e_1\, e_2 : \tau'} \quad (\text{GApp2})$$

Figure 2.4: Original presentation of the static semantics of the GTLC. This has subsequently been expanded to ever more complex type systems. Γ is a mapping from variables to optional types. Δ assigns types to constants. The interesting cases are for application, GApp1 and GApp2. GApp1 handles the case where the function type is unknown, whereas GApp2 deals with cases where the argument type in unknown, but consistent with, the function's parameter type.

$$\overline{T \sim T} \qquad \overline{T \sim \star} \qquad \overline{\star \sim T}$$

$$\frac{T_{11} \sim T_{21} \qquad T_{12} \sim T_{22}}{T_{11} \to T_{12} \sim T_{21} \to T_{22}}$$

Figure 2.5: Type consistency for the GTLC. T ranges over concrete types, such as `Int` or `Bool`.

consistency therefore acts as a replacement for type equality in a gradual system. For instance, the function type Int $\to$ Bool is obviously consistent with itself, i.e. (Int $\to$ Bool) $\sim$ (Int $\to$ Bool), but also with $\star \to$ Bool and Int $\to \star$ and, of course, $\star \to \star$. It would not be consistent with $\star$ alone or Int $\to \star \to$ Int. Any program accepted, or rejected, by the underlying, non-gradual static type system must be treated likewise by the gradual system.

Consistency extends type equality in a conservative manner, allowing the static semantics to be more permissive in the presence of the dynamic type. Applying consistency to increasingly complex static type systems has resulting in its definition becoming more challenging. Consistency has not been consistent since first proposed. In the original Siek system, it is aligned with the mathematics of partial functions, i.e. two partial functions may be considered consistent when every element in the domain of both functions is mapped to the same result.

Figure 2.6 shows a typical presentation of lifting function application in the simply typed λ-calculus to its counterpart in the gradually typed λ-calculus. Note that there are now two rules, rather than one, and type equality has been replaced with type consistency. Subsequent work by Siek *et al.* [179] introduced a pattern matching operator, $\triangleright$, that ascertains whether the expression in function position

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\, e_2 : \tau'} \quad \Longrightarrow \quad \begin{array}{c} \dfrac{\Gamma \vdash_G e_1 : \star \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash_G e_1\, e_2 : \star} \\ \\ \dfrac{\begin{array}{c}\Gamma \vdash_G e_1 : \tau \to \tau' \\ \Gamma \vdash_G e_2 : \tau_2 \qquad \tau \sim \tau_2\end{array}}{\Gamma \vdash_G e_1\, e_2 : \tau'} \end{array}$$

Figure 2.6: Standard static semantics of function application in the simply typed λ-calculus and its gradual counterpart. The gradual counterpart now has two rules rather that one. This is to handle the case where either the expression in function position or argument position is of the $\star$ type.

is of an appropriate shape.

The additional complexity of the semantics in Figure 2.6 is of interest. It is more complex than either the statically typed language would be, or the dynamically typed. As pointed out by New and Ahmed [145], even for such as small language as the STLC, the operational semantics are complicated. Typed calculi have reduction rules for each elimination form; dynamic languages add a partiality to these forms (the possibility of a type error). The semantics here are not modular in the same fashion, and involve comparisons (made at runtime) between arbitrary types. Such comparisons can be arbitrarily hard and have an effect on performance, as discussed in Section 2.2.5.

In subsequent work on consistent subtyping in objects, Siek and Taha [173] reconceived consistency in terms of a restriction operator that states that two objects can be regarded as consistent if their known parts are equivalent or consistent, i.e. $\sigma \lesssim \tau \equiv \sigma|_\tau <: \tau|_\sigma$.

A gradual type system then is a *superset* of the static and dynamic. According to Siek *et al.* [179], a gradually type λ-calculus (GTLC) should encompass both the dynamically typed (DTLC) and statically typed (STLC) λ-calculi. They put forward two theorems which require that the dynamic semantics of DTLC coincide with the STLC when fully annotated. They define static as being the absence of the unknown type. Let $\vdash_S$ be the typing judgement for STLC and $\Downarrow_S$ be the evaluation function for the STLC.

Theorem 2.2.1 (Equivalence to the STLC for fully annotated terms.)**.** *Suppose e is a term of DTLC.*
1. $\vdash_S e : T$ *if and only if* $\vdash e : T$
2. $e \Downarrow_S v$ *if and only if* $e \Downarrow v$

The relationship between DTLC and GTLC is a little more nuanced. The application of *true* to a function $inc : Int \rightarrow Int$ is statically accepted by DTLC, as the language may in fact be considered unityped [170], whereas it should fail to type check in GTLC. Let $\Downarrow_D$ denote the evaluation function for DTLC.

Theorem 2.2.2 (Embedding of DTLC in GTLC.)**.** *Suppose that e is a term of DTLC.*
1. $\vdash \lceil e \rceil : \star$
2. $e \Downarrow_D v$ *if and only if* $\lceil e \rceil \Downarrow \lceil v \rceil$

where $\lceil \cdot \rceil$ lifts constants to the unknown type, thereby 'obscuring' type information.

2.2.1 Cast Calculus

The dynamic type can safely be a 'wildcard' in the static semantics: it is a placeholder which represents a plausible concrete type. We might legitimately co-opt a term from Ahmed *et al.* [4] when they refer to the *Jack-of-all-trades* property of the

$\star$ type. This treatment of $\star$ cannot safely be transported to the dynamic semantics, where we need concrete type information. A *cast calculus* is the most usual method for achieving this runtime type safety. It is an intermediate language with explicit type casts. When an expression e with a *target type*τ, $\langle \tau \rangle e$ evaluates to a value v, the cast checks that the type of v, τ' is consistent with τ, that is, $\tau' \sim \tau$. If it is not, then a error condition is produced. This error condition can come with additional information in the form of *blame*. Figure 2.8.

The *cast insertion judgement* has the form $\Gamma \vdash e \Rightarrow e' : \tau$ which may be read as the expression e is castable to expression e' of type τ in context Γ. Figure 2.7 shows the cast insertions for function application. Rule CAPP1 handles the case where the expression in function position, e_1 is of the $\star$ type. The type of e_1 is cast from $\star$ to $\tau_2 \rightarrow \star$, τ_2 being the type of its argument, ensuring that it is of a function type. CAPP2 handles the case where e_1 has a known (function) type, but its argument is consistent with, but not the same as, its formal parameter. Finally, Rule CAPP3 is equivalent to the 'standard' elimination rule for the function type. The checking for consistency between two types is a non-trivial operation for complex type systems, its effect will be examined more closely in Section 2.2.5.

The *Blame Calculus* is an extension of the cast calculus to include a notion of tracking responsibility for an error. Blame has its origins in the *contracts* of Findler and Felleisen [82] and were first discussed in the context of gradual types by Wadler and Findler [212]. Blame assignment is an active area of research in its own right and is out of the scope of this paper. A gradual type system does not, of necessity, have to have blame, but practical implementations often do. Reticulated Python, of Vitousek *et al.* [208] is a gradual Python with a cast calculus and blame. The gradual security type system of Disney and Flanagan [71], which we shall examine in Section 2.3 also has blame. Blames *labels* attach to each cast. Blame

$$\frac{\Gamma \vdash e_1 \Rightarrow e_1' : \star \qquad \Gamma \vdash e_2 \Rightarrow e_2' : \tau_2}{\Gamma \vdash e_1\, e_2 \Rightarrow (\langle \tau_2 \rightarrow \star \rangle e_1') e_2 : \star} \quad (\text{CAPP1})$$

$$\frac{\Gamma \vdash e_1 \Rightarrow e_1' : \tau \rightarrow \tau' \qquad \Gamma \vdash e_2 \Rightarrow e_2' : \tau_2 \qquad \tau_2 \neq \tau \qquad \tau_2 \sim \tau}{\Gamma \vdash e_1\, e_2 \Rightarrow e_1'(\langle \tau \rangle e_2') : \tau'} \quad (\text{CAPP2})$$

$$\frac{\Gamma \vdash e_1 \Rightarrow e_1' : \tau \rightarrow \tau' \qquad \Gamma \vdash e_2 \Rightarrow e_2' : \tau}{\Gamma \vdash e1\, e_2 \Rightarrow e_1'\, e_2' : \tau'} \quad (\text{CAPP3})$$

Figure 2.7: Function application in the cast calculus for the STLC. Again, it is simple to see the increase in complexity brought about by the sound inclusion of a $\star$ type: what is one rule in STLC is here become three.

can be either *positive* or *negative*: positive blame results when the term within the cast is at fault; negative blame occurs when the context containing the cast is the problem. Figure 2.8 shows an example of a cast, in a refinement type system, which fails with blame. The example comes from Wadler and Findler [212].

They prove the *Blame Theorem*, this states that well-typed terms cannot be the cause of a type error: it characterises safe from unsafe casts. The language they consider contains two forms of cast operation, one which makes type more precise $\star \Rightarrow A$ and those which make the type less precise $A \Rightarrow \star$. The former cannot give rise to *negative* blame: if there is an error, it is in the attempt to go some concrete type A and not in the context. Alternatively, the latter cannot give rise to *positive* blame: a cast to $\star$ never fails, so if there is an error, it must be in the context.

$$
\begin{aligned}
& \langle \mathbb{N} \Leftarrow \mathbb{Z} \rangle^{pn} (-2) \\
\longrightarrow & \\
& \text{if } -2 \geq 0 \text{ then } -2_{\mathbb{N}} \text{ else } \textbf{blame } p \\
\longrightarrow & \\
& \textbf{blame } p
\end{aligned}
$$

Figure 2.8: A failed cast with positive blame. -2 cannot be cast from $\mathbb{Z}$ to $\mathbb{N}$ so the result is the blame label p. This label can contain information useful to the programmer for locating the exact origin of the error.

2.2.2 Gradual Guarantee

The point of all this development is a 'smooth transition', in either direction, between static and dynamic typing.

It is difficult to measure the smoothness of a gradual system's transitions without a formal definition of smooth transition. In an attempt to capture this notion of 'smooth transition', Siek *et al.* [180] propose the *Gradual Guarantee*. The gradual guarantee has proved to be a popular foundation for measuring gradual type systems, but it is not without its problems: it has proved to be very hard to satisfy for complex type systems [110]. The gradual guarantee is a statement of the fundamental intuition behind gradual typing: a developer should be able to modify the type information of programs and have the semantics change in a *predictable* way. In effect this means that making types more statically precise leads to either more runtime type checking (until the program is completely statically typed of course) or rejecting an ill-typed program; otherwise the program behaves in exactly the same way.

$\boxed{\tau \sqsubseteq \tau}$

$$\frac{}{\tau \sqsubseteq \star} \qquad \frac{}{B \sqsubseteq B} \qquad \frac{\tau_1 \sqsubseteq \tau_3 \qquad \tau_2 \sqsubseteq \tau_4}{\tau_1 \to \tau_2 \qquad \tau_3 \sqsubseteq \tau_4}$$

$\boxed{e \sqsubseteq e}$

$$\frac{}{k \sqsubseteq k} \quad \frac{}{x \sqsubseteq x} \quad \frac{\tau_1 \sqsubseteq \tau_2 \qquad e_1 \sqsubseteq e_2}{\lambda x : \tau_1 . e_1 \sqsubseteq \lambda x : \tau_2 . e_2} \quad \frac{e_1 \sqsubseteq e_2 \qquad e_1' \sqsubseteq e_2'}{(e_1\ e_1')^{\ell} \sqsubseteq (e_2\ e_2')^{\ell}}$$

Figure 2.9: Type and Term Precision for GTLC. B ranges over base types. Type precision is the same as the 'naive' subtyping of Wadler and Findler [212].

Given a type precision operator, $\sqsubseteq$, with increased precision to the left (i.e. $e \sqsubseteq e'$ means that term e is more precisely typed than e') and suitable definitions for term and type precision (Figure 2.9):

Theorem 2.2.3. *[Gradual Guarantee] Suppose* $e \sqsubseteq e'$ *and* $\vdash e : \tau$

(1) $e' : \tau'$ *and* $\tau \sqsubseteq \tau'$.

(2) if $e \Downarrow v$, *then* $e' \Downarrow v'$ *and* $v \sqsubseteq v'$.

if $e \Uparrow$ *then* $e' \Uparrow$.

(3) if $e' \Downarrow v'$, *then* $e \Downarrow v$ *where* $v \sqsubseteq v'$, *or* $e \Downarrow blame_T\ l$.

if $e' \Uparrow$, *then* $e \Downarrow blame_T\ l$.

where $\Downarrow$ indicates that the expression takes a reduction step in accordance with the reduction rules and $\Uparrow$ indicates that evaluation diverges.

A language which meets the requirements of the Gradual Guarantee allows the user to safely vary the precision of type annotations without varying the semantics of the program, *modulo* blame on the left-hand-side of the precision operator. In other words, removing annotations should not make a poorly typed program run properly, and adding annotations should not alter the behaviour of the program unless the annotation itself is incorrect, in which case it will result in a trapped type error, shown in Theorem 2.2.3 as $blame_T$.

2.2.3 Abstracting Gradual Typing

We have already mentioned the two key modern approaches to sound gradual typing: those inspired by Siek and alternatively by Felleisen. Neither gives clear a indication on how to *generate* a sound gradual system from a static one. 'Abstracting Gradual Typing', or AGT, offers such a methodology.

Inspired by earlier work on gradual effects by Bañados *et al.* [169], Garcia *et al.* [90] introduce AGT. This is an approach built upon methods derived from abstract interpretation [61]. It relies on the observation that gradual typing can be viewed as a theory of imprecise type information. Type systems themselves have been recognised as an abstract interpretation of the runtime semantics of a program for some time [60].

In AGT, the $\star$ type is given meaning through the two mappings of the Galois connection, α and γ, where α is the *abstraction* function, mapping a point in the concrete lattice to a point in the abstract lattice and its inverse, γ, mapping a point in the abstract lattice to the concrete. The *meaning* of $\star$ is, therefore, the set of static types which it can represent. In the case of AGT, the concrete lattice is the powerset of all types, ordered by set inclusion, for example Figure 2.10. Appropriate definitions of α and γ make it possible to derive systematically both the static and dynamic semantics of a gradual language. AGT contrasts with the

methodology of Cimini [52, 51], which presents an algorithm for automatically deriving a gradually-typed system from an existing static system, which is proved to satisfy the Gradual Guarantee (Section 2.2.2). This algorithm, the *Gradualizer*, generates a gradual type system from an existing, well-formed, static type system and also generates the compiler for the *cast calculus* (Section 2.2.1). The gradualizer has so far only been tested on the simply typed λ-calculus and its extensions. Whether the methodology can be extended to include type systems with such features as parametric polymorphism is an interesting challenge and remains to be seen.

A pleasing property of the AGT method is that any system so derived satisfies, by construction, the first part of the *gradual guarantee* (Theorem 2.2.3). AGT also shows that, at least for a simple functional language, such a method produces a language which also satisfies the dynamic semantic components of the gradual guarantee (parts 2 and 3).

The static semantics of AGT-derived languages strongly resemble those of Siek and others. Of greater note, however, are the dynamic semantics, which rely on real time typing derivations on intrinsically typed terms [158] to enforce type safety. There is no cast calculus in this method, just the evaluation of 'evidence' during execution. As far as known, there are no real world implementations of gradual type systems based on the AGT method, so it is as yet impossible to assess the performance characteristics of its intrinsically typed dynamic semantics. There is no reason to assume that dynamic evidence evaluation should be substantially more efficient than cast insertion. Runtime type errors signal themselves due to the failure to combine evidences successfully. This is essentially a unification problem.

Using AGT, Lehmann and Tanter [122] have systematically derived a gradual

$$\alpha(\text{Int}) = \text{Int}$$
$$\alpha(\text{Bool}) = \text{Bool}$$
$$\alpha(\overline{T_{i1} \to T_{i2}}) = \alpha(\overline{T_{i1}}) \to \alpha(\overline{T_{i2}})$$
$$\alpha\, \emptyset = \text{undefined}$$
$$\alpha(\widehat{T}) = \star \text{ otherwise}$$

$$\gamma(\text{Int}) = \{\ \text{Int}\ \}$$
$$\gamma(\text{Bool}) = \{\ \text{Bool}\ \}$$
$$\gamma(\tilde{T}_1 \to \tilde{T}_2) = \gamma(\tilde{T}_1) \widehat{\to} \gamma(\tilde{T}_2)$$
$$\gamma(\star) = \text{TYPE}$$

Figure 2.10: α and γ functions for STLC with Int, Bool and function application as base types. The wide hat over a metavariable signifies the collecting semantics on that metavariable. An imprecisely typed function such as $f : Int \to \star$ maps to the set of all functions from *Int* to any type, whereas an *Int* in the abstract lattice maps to the concrete *Int*.

refinement type counterpart to a static refinement type system. Creating a gradual system for refinement types poses some unique challenges, not least is, converse to normal practice in gradual systems, it is not desirable to have an unknown formula stand for *any* arbitrary formula in the static semantics. If this were the case the type checker would accept too many programs; anything can be proved *ex falso*. Their type system is a conservative extension of a static type system which preserves the typeability of less precise programs.

2.2.4 Extended Type Systems

Sergey and Clarke [171] explore the gradualisation of ownership types. Ownership types in object-oriented programming can enforce some extremely useful properties, such as effective memory management and confinement properties. Gradual typing has also been extended to effects [168], generics [111] and session types [200].

Type inference for gradual typing has been explored by both Siek and Vachharajani [178] and Garcia and Cimini [89]. While they both take different approaches to the problem, they result in type checkers which accept the same programs statically. Siek's type inference algorithm, however, is special purpose, and there is no obvious way to extend it to other type systems. This is similar to the problems noted earlier on extending consistency. Gradual typing has been applied, at least in part, to parametric polymorphism.

Igarashi *et al.* [110] present a gradualised *System F*. They prove the gradual guarantee for the static semantics, but provide only an outline for an approach to proving the dynamic element of the gradual guarantee. Ahmed *et al.* [5] take a slightly different approach when they examine the polymorphic blame calculus in a gradual setting. Gradual typing has even been extended, at least in part, to the world of dependent types. Ou *et al.* [149] show how a form of dynamic typing can be embedded within a dependently typed language, but consider only a core language. Tanter and Tabareau [195] introduce a form of gradual typing for *Coq* [196]. Their method is primarily concerned with the *safe* postponement of proof terms, using user-inserted casts. Rather than gradualising elements of the signature (the proposition), the proof is gradualised.

A theme running throughout research with respect to richer type systems is the complexity of proving this second, *dynamic*, part of the gradual guarantee [145]. Moreover, there is very limited guidance on *how* to design a language which satisfies this dynamic element of the guarantee. Chung *et al.* [50] observe that various different approaches to gradual typing have different ideas about what should constitute an error.

No language used in production, although going under the moniker of gradual typing, satisfies the gradual guarantee; many do not have the properties

of a sound type system. The most notable case of this is TYPESCRIPT. Swamy *et al.* [191] propose *TS*⋆, a sound gradual type system and compiler, for a core of JAVASCRIPT. *TS*⋆ is an interesting addition to the literature in that it attempts to address the unsoundness of TYPESCRIPT's type system, and also adds a security mechanism in the form of a base type '*un*' which is used to label potentially malicious JAVASCRIPT code. The aim of this security mechanism is to prevent certain classes of common attack; it is not intended to enforce noninterference (Section 2.1.1). *un* is a first class type in *TS*⋆, it may be used anywhere in code, including within records and in returned results. Wrappers ensure a strict memory separation between *un* and the rest of *TS*⋆.

2.2.5 Gradual Typing Efficiency

As already noted in Section 3.1, sound gradual typing has seen less industry adoption than *optional* typing. While no systematic study has been conducted to explore why this might be the case, one obvious barrier is the relative *inefficiency* of cast-based languages.

Some implementations have been examined and shown to suffer from performance degradation at runtime [157, 208]. Takikawa *et al.* [193] claim that the cost of Typed Racket's soundness is not tolerable. Typed Racket is a sound gradual scheme. Their experimental approach is to take an untyped program, measure its performance characteristics, and then gradually type that program, creating a lattice of programs ordered by type annotation precision and then measuring their performance. They build a performance lattice which can then be used for analysis of runtime characteristics.

In general, they found that fully typed code runs slightly faster than completely untyped code, but for all points in between, performance degradation is notable, sometimes as much as 90 times slower than the untyped version. It is

Language	**Performance**
TypeScript	-
StrongScript	1.1x
GradualTalk	5x
Typed Racket	121x
Reticulated Python	
Transient	10x
Monotonic	27x
Guarded	21x

Table 2.1: Self-reported performance pathology as given in Chung *et al.* [50]. TYPESCRIPT, being an *optionally typed* language, does not suffer from runtime performance degradation, but likewise does not offer runtime type safety.

unclear whether these performance costs are a necessary element of soundness guarantees or are an artefact of Typed Racket's implementation. Certainly, such a performance cost is likely to mitigate again widespread adoption, as the gain from extra type safety is unlikely to be sufficient compensation for the performance loss. Even with few annotations, the slowdown in execution is surprising. Table 2.1 gives a breakdown of some languages, both optional and gradual, with self-reported performance degradation due to casting.

Obviously, if annotations are erased, as in TypeScript, then there is no performance cost. Unfortunately, in the case of security labelling, dynamic monitoring in the presence of gradual typing would be a necessity, as no existing, non-IFC, language runtime engine has information of policy infringements. Potentially, given that IFC is likely to be of importance in areas where speed is often a re-

quirement (for example in online financial transactions), it is vital that the cost of security is not paid for in performance.

Muehlboeck and Tate [139] argue that many of the performance problems reported for sound gradual typing can be addressed via careful development of a type system at the same time as its implementation. They present a nominally-typed object-orientated language with minimal performance overhead. This work of course does not help to address the problem of fitting existing languages with efficient gradual typing. Bauman *et al.* [23] also address the problem of performance, but instead focus on the use of *just-in-time* compilation to significantly reduce performance overhead. They present *Pycket*, a Jit compiler for Racket. This allows them to compare their results directly with the results presented for Racket by Takikawa *et al.* [193]. They report results of 90% reduction in dynamic overhead. Thus, with dedicated engineering effort and appropriate language design, it might be possible to get the performance hit down even further.

Kuhlenschmidt *et al.* [120] use *ahead of time* compilation and space efficient *coercions* [104, 175] to implement runtime casts. Furthermore they use the *monotonic references* of Siek *et al.* [181] to reduce overhead in statically typed code (Table 2.1). *Monotonic reference* semantics update the type of heap objects *in place*: replacing $\star$ with a more precise type. This update is then propagated recursively through the heap. Casting can only cause a heap cell to become more precise.

2.2.6 Static and Dynamic IFC

We have seen that statically enforced type systems guarantee noninterference at the expense of being highly restrictive: they reject too many secure programs [162]. An alternative to the static approach is to shift the burden of IFC enforcement to purely dynamic methods. This has a great advantage of allowing for systems of greater flexibility: Sabelfeld and Russo [166] prove that purely dynamic enforce-

ment is more permissive than static, at least in the case of flow-insensitivity. It is logical to explore the benefits of hybrid approaches to IFC.

Malecha and Chong [127] develop an IFC type system for a core imperative language which they call $Limp_{\mathscr{L}}$. They show it to be strictly more expressive than the Jif system. The need for this increased expressivity is linked to the necessity of rewriting existing Java code to work in Jif [58]. This *post hoc* typing of existing code is an extremely important use case and is largely ignored by the research literature. Again, gradual typing offers a logical means by which existing code can be securely typed without *necessarily* breaking working projects. Malecha and Chong provide an interesting example from Jif where the information flow is secure but is still rejected by the compiler [127], as in Listing 2.2.

The two variables y and z have different security levels. The program ensures, with a runtime check, that information is allowed to flow from level p to level q. If this flow is allowed, and if y is greater than zero, an exception is thrown. The exception handler then assigns the value of 1 to z. This is allowable, because the runtime check has already ascertained that flow from level p to q is permitted. This is rejected by Jif however because the assignment to z takes place outside of the lexical scope of the dynamic security test $p \sqsubseteq q$. The compiler is not able to keep track of the fact that the runtime check allows for the flow of information from p to q.

The problems associated with dynamic IFC management are quite clear here. It greatly increases implementation complexity and furthermore has an inevitable effect on performance. Monitoring implicit flows is the most difficult aspect of dynamic information flow control. Indeed it has been show that *purely* dynamic mechanisms cannot precisely enforce noninterference [165]. Dynamic methods can however approximate, in a safe manner, a permissive form of noninterference.

```
int {*p} y = ...;    /* y is protected by label p */
int {*q} z = 0;      /* z is proteced by label q */

try {
    if (p ⊑ q) {
        /* information can flow from y to z */
        if (y > 0) throw new Exception();
    }
}
catch (Exception e) {
    z = 1;
}
```

Listing 2.2: Secure information flow rejected by Jif: assignment to z is rejected, even though there is no flow, because the compiler has 'forgotten' that flow between p and q is permitted.

A sound gradual type system necessarily has recourse to a form of dynamic monitor or cast calculus, so the problems associated with dynamic IFC will have an effect on what can be described and controlled by a gradual IFC type system.

The thrust of recent research has been towards hybrid systems, mixing static and dynamic enforcement. These can offer the greatest mix of soundness and permissiveness. Russo and Sabelfeld [162] give a detailed discussion of the relative merits of hybrid systems. Chandra and Franz [41] combine static and dynamic techniques for IFC on the JVM, and report that, even for large applications, the performance impact of dynamic monitoring was at worst a 2x slowdown in execution. Austin and Flanagan [16] propose an efficient purely dynamic information flow analysis. The majority of research into dynamic and static

systems has focused on modelling languages in terms of the λ-calculus or *while* languages. There is however a small body of work which explores modelling IFC in terms of the π-calculus. Kobayashi [117] created a new type system form IFC in the π-calculus. Hennessy [103] introduces a security-typed version of the asynchronous π-calculus.

2.2.7 Granularity

Dynamic IFC systems can be divided between *coarse-grained* and *fine-grained* approaches [189, 156]. Coarse-grained models associate security labels with entire modules or objects, rather than at the level of the individual variable or function as is the case in fine-grained IFC (*cf.* Section 2.1.3).

Course-grained IFC works by applying security labels to entire blocks of code and then only monitoring the flow of information between the blocks. The interiors of the blocks themselves are 'black boxes'. This has seen most application in the realm of IFC operating systems [155]. This has the benefit of increased simplicity over the fine-grained approach used by most type system based solutions: it is not necessary to give every variable and function a security label. This might be an extremely useful property when gradually applying a security policy to a program. Coarse guarantees could be progressively refined as necessary.

Stefan *et al.* [189, 190] propose a library-based system, *LIO*, for Haskell which seeks to merge the course- and fine-grained approaches. As with the coarse-grained approach, *LIO* associates a label with a current context, using a monad, also called *LIO*, to restrict computations to a safe subset of Haskell. However *LIO* also allows the user to associate labels with particular values. These labels are typically created at runtime and are used to control dynamic information, such as user input. LIO is a flow-sensitive system.

2.2.8 Other Approaches to Information Security

Many other approaches to securing informationflow have been studied that are not type-based. While this literature review is primarily concerned with type-based IFC, it would be remiss not to mention the most important recent contributions in this area. Joshi *et al.* [113] suggested a semantic approach to secure information-flow, but this has seen relatively little uptake and is not the basis of any practical implementation. Devriese *et al.* [69] propose a novel model, based on *secure multi-execution* (SME), and demonstrate both its soundness and its practicality for certain domains. DeGroef *et al.* [98, 97] have realised this idea and developed a functional web browser, *FlowFox*, which uses SME.

SME relies on the observation that it is possible to run a program again and again for each security level, substituting dummy values for the higher level data. The program is also run at the highest level with the real confidential data, and if all low output states are equivalent, then the program is necessarily secure. *FlowFox* also allows for rewriting of programs under SME so that all programs become noninterferent, with automatic program transformation in the event of an infringement of IFC policy. They analyse the cost of this to be a substantially greater use of memory and CPU, but with little, if any, time penalty. They even show that in certain circumstances performance improves. This makes SME a perfect fit for the browser, but would not be an obviously applicable method for systems programming, where memory consumption and CPU cycles are managed at a micro level.

Unfortunately, in the case where the web page is, for example, a *mash-up*, with n principles corresponding to URL domains, SME may require up to 2^n processes, as Austin and Flanagan have pointed out [19]. This is expensive. They use secure multi-execution as the basis for their *faceted values*, which simulate multiple

```
y_l = false;
if x_h then y_l = true;
```

Listing 2.3: Assignment within a conditional statement.

executions for different security levels. This approach has minimal overhead and avoids the problems of stuck evaluation in previous dynamic approaches. Consider the problem of implicit flows, such as assignment within a conditional, as in Listing 2.3.

They argue that the correct value for y depends on the authority level of the observer. If the observer has high authority, then y has the value `true`, otherwise it should remain as `false`. Faceted values represent this duality with a triple, with k as a principal and two values, V_h and V_l:

$$\langle k \,?\, V_h : V_l \rangle$$

This appears as V_h for viewers of high security, and V_l otherwise. A value which is already public is represented simply by itself, as it has no alternative faces to project to the world. A private value is represented as

$$\langle k \,?\, V \;:\; \bot \rangle$$

They demonstrate the system by developing an idealised language, λ_{facet} which they equip with mutable reference cells, I/O and a mechanism for automatically created faceted values.

2.3 Gradual Information Flow

Now that we have examined both gradual typing and IFC mechanisms using type systems, we can develop the central area of investigation of this survey: gradual typing for information flow control. This is an area in its infancy, but important work has already been done in proving that it is at least plausible. It is not just a question of developing efficient semantics but new problems emerge in the combination of IFC and gradual typing which do not yet have satisfactory solutions.

2.3.1 Gradual Security Type Systems

Disney and Flanagan [71] produced the first paper on gradual security typing in 2011. They consider a simple λ-calculus, enriched with three base types, `Int`, `Bool` and `String`, extended with a gradual IFC system, resulting in a core language which they call λ_{gif}. In λ_{gif} every value and type has an associated security label, which form a security lattice. Unlabelled values are assumed to have the $\perp$ label, making the value public. Conversely, they assume unlabelled types to have the $\top$ label, allowing it to describe values of any security level.

Figure 2.11 details the language syntax: $\Rrightarrow$ is the *labelling* or *stamping* operation, which labels data. `1:Int` $\Rrightarrow$ $\texttt{Int}^H$ stamps the data as confidential, and $\Rightarrow^p$ is the cast operator which checks that flow is permitted. The p superscript is a blame label (Section 2.2.1). The dynamic semantics of λ_{gif} are formalised using big-step semantics. Figure 2.12. At runtime λ_{gif} detects illegal down-casts in order to guarantee termination-insensitive noninterference. The operational semantics for the cast operation is supported by three rules, which check that a runtime label is compatible with a specified static label. A blame label, p, is applied to signal the part of the code which is at fault. They follow the usual distinction between

$i ::= \text{Int} \mid \text{Bool} \mid \text{Str}$ *Base Types*

$a, b ::= i \mid A \to B$ *Raw Types*

$A, B ::= a_k$ *Labelled Types*

$t, s ::= v \mid x \mid t\ s \mid op\ \bar{t} \mid t : A \Rrightarrow B \mid t : A \Rightarrow^p B$ *Terms*

$r ::= c \mid \lambda x : A.t$ *Raw Values*

$v, w ::= r^k$ *Labelled Values*

k, l, m *Labels*

$\Gamma ::= \emptyset \mid \Gamma, x : A$ *Typing Environment*

Figure 2.11: Syntax of λ_{gif}.

Figure 2.12: Some rules from λ_{gif} operational semantics, including cast insertion rules and the *blame* calculus.

positive and negative. Positive blame, p means that the term within the cast is at fault, while negative blame, $\overline{p}$, means that the context of the cast is at fault.

As an example of the E-CAST-FN rule, they consider a function `f` : $\texttt{Int}^l \to \texttt{Bool}^l$. If the codomain is strengthened, via cast, to `f`: $\texttt{Int}^l \to \texttt{Bool}^l \Rightarrow^p \texttt{Int}^l \to \texttt{Bool}^h$, then the new wrapper function created, `f'`, has type `f'`: $\texttt{Int}^l \to \texttt{Bool}^h$. As the original return value of `f` is public, it is always safe, from an IFC perspective, to up-cast the result of f to that of f'. This is directly comparable to the safety of casting from a concrete type to $\star$ in a non-IFC gradual system: such a cast cannot fail. Alternatively, if the domain of f is strengthened to `f'` : $\texttt{Int}^h \to \texttt{Bool}^l$ then the cast from `x'` : $\texttt{Int}^h \Rightarrow^{\overline{p}} \texttt{Int}^l$, needed to pass the argument to the type expected by `f` will fail when `x'` is private.

λ_{gif} provides *Termination Insensitive Noninterference*. The use of blame however can be another covert channel: a program which fails due to a blame assignment

terminates in an error condition, satisfying *Termination Insensitive Noninterference*, though potentially leaking information via the information in the blame label. λ_{gif} is a sophisticated initial offering, but being initial it has a number of limitations. Perhaps most obviously, there is no cast calculus for λ_{gif}, so a developer using this system, or a system derived from it, would have to manually insert casts and labelling operations, rather mitigating against the advantages of gradual typing. From an IFC perspective, the type system is only a little more sophisticated than that of Volpano (Section 2.1.2). There is no scope for declassification in this system, nor is it flow-sensitive. The relationship of λ_{gif} to the gradual guarantee is also unclear.

One finds another gradual security core language in the work of Garcia and Tanter [92]. They gradualise Zdancewic's λ_{SEC} [223] using their AGT method [90] to produce a gradually typed security language, $\lambda_{\widetilde{\text{SEC}}}$. $\lambda_{\widetilde{\text{SEC}}}$ is especially interesting for being the only existing formalisation of a gradual security language that does not require the manual insertion of casts into the source code to provide NI guarantees. The language for which they do this however is very simple indeed, a core calculus with booleans and function types only. This allows them to bypass the problem of gradualising the base types at the same time as the security types. Disney and Flanagan too ignore this problem, applying graduality only to the security types and leaving the base types with a standard static system.

The static semantics for $\lambda_{\widetilde{\text{SEC}}}$ are similar to other examples of gradual typing. Of more note are the dynamic semantics. This relies on the notion of the *interior* of a label. The interior of a label is a developing calculation which tests that a security label is consistent with the requirements of the information flow. The interior starts in the form of *initial evidence*, deriving from information obtained during static type checking. For simple types the interior is easy to calculate,

```
int max(int x, int y) where { x <= ret, y <= ret } {
    if (x <= y) {x = y;} return x;
}
```

Listing 2.4: LJGS syntax with a security polymorphic type signature.

for example a `bool` false^H has the interior $\langle \text{H}, \text{H} \rangle$. Using a consistent transitivity operator $\circ <$ and a *recursive* meet operator $\sqcap$, it is possible to reduce the evidence provided by interiors during evaluation, thereby detecting an infringement of the IFC policy without needing explicit casts.

The most mature presentation is Fennell and Thiemann [78], who propose a gradual IFC type system for a core ML-like language. They prove NI for their language, ML_{GS}, using term erasure. They account for the use of flow-sensitive attacks against languages which contain references [166]. Fennell and Thiemann have also looked at gradual security typing for object orientated languages [79]. They develop a lightweight Java, LJGS. LJGS is a Java subset which excludes threads, exceptions and reflection. Like most IFC systems, LJGS using the lattice model to express permissible information flows. A variable must be labelled either with a fixed security level or marked as to be dynamically checked. Local variables are *flow sensitive*, whereas object field types are treated as *flow insensitive*, in order to keep the system lightweight. LJGS is the first application of gradual security types to something more than a core language. LJGS makes the reasonable assumption that only the public part of the result and of the heap is observable by an attacker. LJGS only gradualises the security component of the language, primitive data types must still be fully declared as per 'normal' Java.

Listing 2.4 shows a simple function with a *security-polymorphic* type signature. As the security levels of `x` and `y` are not bounded by fixed values, `max` may be

```
int matMstDyn{Logger, log int x, int y} where { x <= ret, y <= ret
    , log <=  ⋆ } and { ⋆ } {
    if (x <= y) { x = y; }
    log.dbuf = "max was called"; return x;
}
```

Listing 2.5: Dynamic LJGS with the $\star$ type. This is checked via dynamic monitoring. Note the complexity of the type signature even for this simple function.

called under any program counter with arguments that satisfy the constraints. It may even be called with two dynamic arguments. This polymorphism can also be extended to include a dynamic type, $\star$, which relies on runtime information to resolve, as in Listing 2.5. This polymorphic signature relies on dynamic IFC. Both `x` and `y` can flow into *ret* but `log` has a dynamic security label.

The surface complexity of LJGS and Disney and Flanagan rather mitigates against any advantage the graduality brings to the table for treating IFC typing. Reducing the annotation burden is essential. There is no work at all into the performance characteristics of sound gradual security typing. If its performance characteristics are as bad as those reported for Typed Racket, it is difficult to imagine that it will see much use, especially as those areas which are most a natural fit for IFC (encryption, online shopping and banking) are also those areas where users expect to see a reasonable degree of performance. Chapter 4 proposes a new approach to gradual IFC that solves the refactoring and performance problems of the methods discussed here.

2.3.2 Quantified Information Flow

The study of the quantification of information within software has a small, but active, community. Quantified information flow, or QIF, is a comprehensive mathematical theory to explain precisely what information is [62], what information flow is, and how to make quantitative assessments of flow. Alvim *et al.* [7] provide the best summary of the state of the art in this area. It is not possible to cover here all the multitude of uses and approaches of QIF, so we focus on those most relevant to Chapter 3 and Chapter 4.

Exact quantification is possible in formal systems, such as simple *while* languages. The work of Clark *et al.* explores exact measurement of information in these confines [137, 55, 56, 57]. It has not yet been possible to scale these exact approaches to real world software systems, and the research trend has focused increasingly on estimation of information flow, rather than exact measurement.

Smith [187, 186, 188] gives a comprehensive overview of the various measures used for measuring information flows, with a special emphasis on information security. The most common measures are entropy and mutual information. Generalised gain functions [6] are increasingly used in security orientated information theory, as they more accurately model different attacker models. For example, min-entropy measures the "one shot" risk to data in a way that mutual information simply cannot. We use mutual information throughout this thesis, though the techniques should generalise to other information theoretic measures. Mutual information has the advantage of being useful even without an attacker model, as we demonstrate with ICMs in Chapter 3.

2.3.3 Entropy Estimators

Estimating the entropy of a random variable X is an important problem that has many uses. If one has infinite data, or exact knowledge of the underlying distribution of that data, then no estimation is necessary. It is sufficient to plug the probability vector p into the equation for (continuous) entropy

$$H(p) = -\sum_{k=1}^{K} p_k \log(p_k) \tag{2.1}$$

where $0\log(0) = 0$, and $K \geq 2$ is the number of different outcomes. Being a probability distribution, we require that $p = (p_1, p_2, \cdots, p_K)$, such that $p_k \geq 0$ and $\sum_k p_k = 1.0$.

Infinite data or exact knowledge is rare, so estimation is necessary. Entropy estimators can be classified as working on either *discrete* or *continuous* data. We examine discrete estimators only. This is motivated by our use of hashing in Chapter 3 to discretise the data provided via our observations. Broadly comparable tools, such as LEAKWATCH, make similar decisions.

The simplest possible estimator is the naive *plugin estimator*. This provides an entropy estimator purely from the histogram of observations, that is,

$$\hat{H}_N = -\sum_{k=1}^{K} \hat{p}_k \log(\hat{p}_k) \tag{2.2}$$

where the circumflex above a variable indicates that it is an estimate. Equation (2.2) is equivalent to

$$\hat{H}_N = \log(n) - \frac{1}{n} \sum_{k=1}^{K} h_k \log(h_k) \tag{2.3}$$

where h_k is the sum of outcomes in the histogram. Equation (2.3) is easy to implement and efficient, but has a notable drawback. On average, it underes-

timates the true entropy Equation (2.1) [21]. Informally, we have the following situation. Consider a single bin, k in our histogram h for a random variable X. To know the precise contribution of k to the $H(X)$, we need the precise probability of k, p_k. Not having this, we substitute the frequency count instead, $\hat{p}_k = h_k/n$, where n is the sum of all $k \in h$. The marginal distribution of $\hat{p}_k$ is a binomial distribution. Evaluating the *concave* function $f(x) = -x\log(x)$ leads to the underestimation of entropy *on average* due to Jensen's inequality.

Improved estimators take this bias into account. The simplest is the Miller correction [150], proposed in 1955. This works by adding a constant offset, K in the bias expression (Equation (2.4)). The Miller-Madow algorithm extends this by also estimating K from the data.

$$\hat{H}_M = \hat{H}_N \frac{K-1}{2n} \tag{2.4}$$

Grassberger proposes a mutual information estimator in [118]. Grassberger's estimator is computationally efficient and is generally considered both robust and useful. Indeed, RIFFLER (Chapter 3) falls back on Grassberger when there is insufficient sampling to use the Nemenman-Shafee-Bialek (NSB) estimator. We shall examine this in more detail in Section 4.5.

Chapter 3

Ranked Information Flow

Information flows through all software. Some of this information should be secure, some not. Problems arise when information flows in unintended ways. These are not just security problems, but general software engineering problems. Flow-aware tools exist but have had little impact on development practice to date. A significant barrier to the adoption of these tools is the level of expertise in information theory required by the programmer. We propose to lower this bar through enabling a more exploratory, feedback driven approach.

We present *information flow cartography*, a means to map and visualise the flow properties of software. Our approach produces topographical heat-maps that allow a programmer to readily understand the internal information flow of software under development. We instantiate our approach with RIFFLER, a tool for creating a flow cartography. This automatically creates information flow maps via a lightweight program transformation. It requires no specialist knowledge to use. It is tunable and allows the user to produce maps at any level of detail. Unlike other approaches, such as dynamic taint analysis, RIFFLER requires no initial assumptions about what is important in the code.

We show that RIFFLER produces an information contour map that can usefully approximate brute-force flow estimation methods after just 2 hours of fuzzing. RIFFLER uses *Ranked Information Flow* (RIF), a novel, practical, feasibly computable abstraction of information flow. By contrast, existing information theoretic approaches to flow quantity analysis rely on an expensively large number of observations or formal methods with limited scalability in order to draw conclusions. RIF requires 60% fewer samples than the minimum number required by more precise estimation tools. We evaluate RIFFLER via case studies that demonstrate our approach. These include use of flow contour maps to both find security problems in code and aid code navigation and comprehension.

"We cannot get anything for nothing, not even an observation." Dennis Gabor [86]

3.1 Introduction

Programs are information processing systems, but we rarely have any idea how information is moving around inside them. Knowing how information moves can tell us whether it is moving correctly and securely. Knowing where there are large changes in information flow reveals where import control flow decisions are being made: moving from high to low flow shows "code funnels", where a larger input space is partitioned. Information theory (IT) [172], specifically *Quantitative Information Flow* or QIF [188], allows us to measure flow, but has barriers to practical use.

Assuming we overcome these barriers, we can build an information map of a program. Amongst other uses, this map can be used to consider the security policy of the program. A security policy defines what information can be read

and written by different classes of users. It is not easy to know if a program satisfies a particular policy, but it is also not always easy to know what a policy should be: program interactions are complicated. One might mark a variable as private, and another as public, but such annotations might lead to unexpected problems via transitivity of *information flow*. This problem has familiar siblings for every developer, where data no longer has the expected form due to preceding calculations.

There are many tools that track information flow through a program, with an eye to security (Section 3.7). They all rely on a least one of the following assumptions:

1. the pre-development existence of a security policy: the developer knows what is secret and what is public;
2. knowledge of the attacker model: the powers of the attacker, whether they can access memory, read the source code, or provide inputs *etc*;
3. declassification (*i.e.* how much information can be allowed to leak from a program.)

It is common for developers to lack this knowledge and be unable to exploit the power of information flow control development techniques.

There are two principal barriers to building these maps: knowing what to observe, and making enough observations to draw conclusions. The first is that it is very hard for a developer to know what is important when there is no explicit policy. One question is: “What do I need to track?”. Another is: “Have I done enough work to make an estimate of what I am tracking?”. The computational complexity of many QIF calculations is PSPACE-complete and, in general, a large number of observations is required (Section 3.3.1) for a precise QIF estimate.

To solve the first problem, we introduce a testability transformation [101] that uses sensible defaults to track input/output at the function level. We introduce Ranked Information Flow, or RIF (Section 3.3), to address the second problem. RIF approximates the relative *rank* of flows to ameliorate observation under-sampling and mitigate the cost of more accurate estimation. Leveraging RIF, we also propose the *Information Contour Map* (ICM) to help non-specialist developers gain knowledge about flows in their program. RIF abstracts dynamic QIF, improving computational feasibility. The ICM brings this information to the non specialist, without burdensome detail. A developer can use this map to explore the information flow semantics of the program, confirming or denying intuitions about information policy. The RIF information map does not, by default, offer strong statistical guarantees, but it also does not require the assumptions of other security tools. In particular, an ICM can help identify malware (Section 3.2).

The assumption for existing QIF tools (Section 3.7) is that tight precision or a tight bound is always the goal. Precision is what we want when we know precisely where to dig, but more high-level mapping is what we require when we are exploring a landscape. Knowing in intimate detail one small square of land does not help us put that small plot in context. RIF creates that context, yet still allows for the more precise details. We use RIF to map information flows over a program's callgraph and rank them via *FlowForward* (Section 3.3.2), a new information-theoretic measure that normalises mutual information (MI) by the size of the input.

We instantiate *FlowForward* and RIF in RIFFLER[1], a tool to perform information theoretic code cartography. RIFFLER uses a testability transformation to gather

[1]A *riffler* is a small, double ended file for detailed, precision work. Our tool uses RIF to enable information-theoretic code exploration.

program information from a normal testing regime (Section 3.4). RIFFLER's scope is tunable along two dimensions: scope of observation and observation detail. RIFFLER comes with an annotation system that allows the users to add or remove annotations with minimal user intervention (Section 3.4.2). This annotation system bootstraps at the function level as a default, but additional annotations can be manually added or removed.

Our principal contributions are:

- We introduce IT cartography, a means to explore the security policy a program implements (Section 3.2);
- We introduce the concept of RIF and also *FlowForward*, an IT measure which can be meaningfully ranked (Section 3.3);
- We present RIFFLER, a security cartography tool integrated into normal testing that uses RIF to map information flows over a program's callgraph (Section 3.4).
- We show that RIF is computable and efficient (Section 3.5);

RIFFLER's implementation, and the scripts and data used to evaluate it, can be found at https://github.com/kellino/python-riffler

3.2 Motivating Example

Every piece of software has a security policy. For something like a calculator, for example, the policy might simply be that no data is secret. Other programs, like login applications, might have a policy that very little information about the password is revealed, *i.e.* just success or fail. Other programs might have considerably more subtle policies. Knowing what the policy should be is not the

same as knowing that the policy is satisfied. In order to discover a program's security policy as instantiated, a map of what a program is doing would be extremely useful, especially as a formal definition of policy is usually lacking.

RIF, and our tool, RIFFLER (Section 3.4) allow developers who lack training in information theory and information security to leverage their general programming knowledge to explore hypotheses about what flows should and should not be. In particular, anomalous flows become much more obvious. The interpretation of flows, and whether they are anomalous or not, depends on context.

If we look at the left-hand side of Figure 3.1, we see the ICM of a small PYTHON program. The ICM is a colour coded extract from the program callgraph; it shows how much information is passing through each function. This transit is *FlowForward* (Section 3.3.2). This extract can be sliced from the larger callgraph by looking for large changes in information flow. This large change occurs between `__init__` and `log`. Given just a callgraph, `log` does not warrant any special investigation, but taken together with its *FlowForward*, we ask why is it only called *here* in the callgraph, and why does it appear to do nothing with its information? The program does not keep any log file. The `log` function in Figure 3.1 is clearly different from all the other functions in the graph.

Let us add more fine-grained annotations. By default, our tool, RIFFLER, records at the function level, but it also has some heuristics annotations at the statement level (*i.e.* capturing print statements). We requested more detail from RIFFLER and produced the map on the right of Figure 3.1. The `log` function has gone from 0 to 1, but there is still no evidence of a log file anywhere. This is deeply suspicious. Closer inspection of the function reveals total information exfiltration. Our private details are being sent, unencrypted, over the internet. The only physical code examination required was at the end of the process, to see

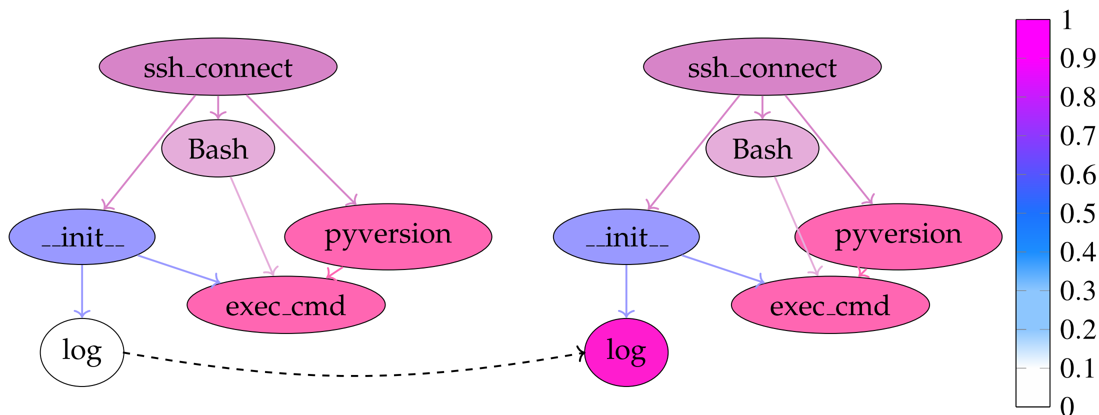

Figure 3.1: The left shows an extract of the callgraph from ssh_decorator with default RIFFLER settings. Immediately, one can see that `log` is both in an unusual position in the callgraph, and is an information "black hole"; it appears to do nothing with the information sent to it. The program keeps no obvious log file. It is clear that `log` warrants further investigation. The extract on the right shows the same callgraph with manual annotations added to `log`. This function is not losing any information at all; this is suspicious, as we might expect a logging function to compress some of its input into formulaic messages.

exactly what was being stolen.

This `log` function comes from the module *ssh-decorate*, which was "back-doored" on PyPI by insertion of code that sends a user's private credentials to an external url. These credentials were sent without encryption. While the offending code is no longer available through PyPI, and the GitHub repository [11] never contained the offending code, it is possible to reconstruct the malware from code snippets online [40] and the code on GitHub. We mocked a server and tested *ssh-decorate* using fake credentials and our tool, RIFFLER.

3.3 RIF: Ranked Information Flow

We present ranked information flow as a feasibly computable abstraction of QIF. Traditional dynamic QIF suffers from high sampling requirements (Section 3.3.1). IT measures, such as mutual information (MI), can have misleading results if ranked naïvely, so we introduce a new measure, *FlowForward* (Section 3.3.2). Ranking directly by *FlowForward*, or any IT measure, can produce too detailed a picture, so we also show how to simplify the *FlowForward* to produce an abstract map of program information flow (Section 3.3.5).

3.3.1 Cost of Dynamic QIF

QIF is a powerful tool for measuring security leaks in a program. Many tools exist, that differ in their scalability [26] and accuracy. None of these tools has made much impact in software engineering. The reasons vary per tool, but some combination of heavy resource consumption, expensive calculation, or requiring a security policy, *i.e.* knowing what to observe and what to ignore, always comes into play.

Approaches that rely on dynamic observation, in particular, suffer either from under-sampling or high resource consumption. To demonstrate this, we took a simple Java program that takes an integer and appends it to increasing amounts of "noise". We limited the integer value `binary` to either 0 or 1. IT says that `binary` contains 1 bit of information. We generated noise (random numbers) of a given length and appended the `binary` value, as a string, to the string of noise. We measured the *mutual information* (Equation (3.1)) between the two; mutual information being a measure of dependence between two variables. To measure the mutual information, we used LEAKWATCH [49]. This is a high quality tool for point-to-point IT analysis in Java. It relies on heuristics to determine when

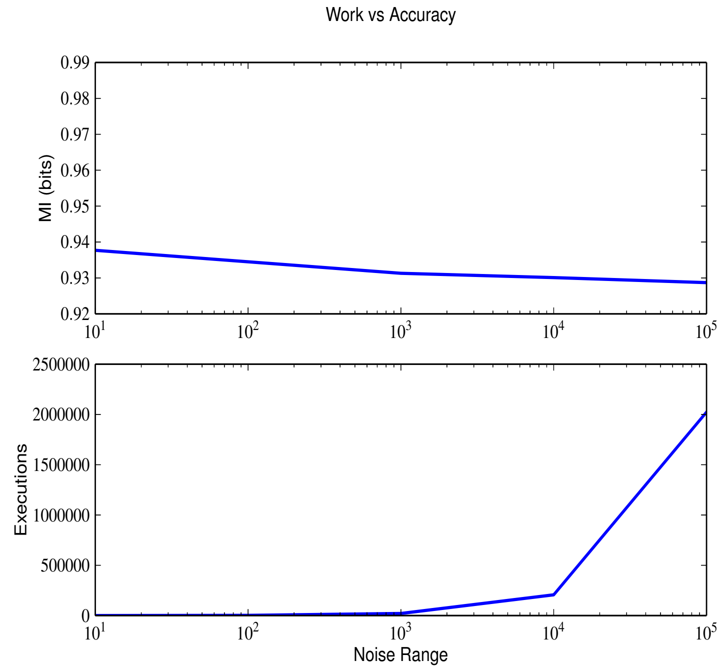

Figure 3.2: The effect of population size on the amount of work required to estimate MI is very large. As the population size increases, the number of samples required for an estimate increases linearly. Using sophisticated entropy estimators (Section 3.3.4) and ranking (Section 3.3.5) reduces the amount of work required.

the minimum amount of samples has been taken[2]. We ran the program and counted the number of executions required before reaching the default statistical threshold.

Figure 3.2 shows that the amount of work increases markedly with the size of the observation space. The MI estimate, however, only decreases a little. This shows two things: MI is a robust measure of dependence between two variables, and it is just not practical to estimate MI in this fashion. This is just

[2]The full details can be found in the paper, but there must be at least 4 times the product of unique input and output observations.

one point-to-point observation, yet takes hours to run. For a string 6 characters long, we required $2,021,438$ executions before achieving confidence, but in our experiments, the most any one program was sampled was $404,682$ (Section 3.5).

We cannot defeat fundamental complexity. The big-O complexity of quantitative information flow (QIF) calculations is high. The QIF bounding problem, checking whether the flow in a program can be bounded above by a constant d, is PSPACE-complete [43]. Other problems in the QIF space have a similar complexity [44].

We want to map information as it moves through a program. Using standard QIF directly for all program variables is too expensive. We present RIF as a means to create a scalable, useful, approximation of information flow over an entire program, and not just over a few select variables. We assume that we do not, *a priori*, know whether a vulnerability exists (though we might know what we want to keep secure). As we do not know, it is unwise to make assumptions about how many observations an attacker can make. These considerations motivate an information theoretic measure which is not primarily concerned with one-shot "guessability", that is normalised in such a way as to allow for easy comparison, and that provides reasonable results even when resources are limited.

A map of an entire system should not be overwhelmed in detail. This motivates our use of ranking. Knowing the relative magnitudes of flows within the callgraph provides sufficient information to describe that system. Once the system has been mapped, and nodes of interest identified, one can use target those nodes for more detailed cartography.

RIF solves the cost problem of QIF by recognising that relative magnitude is more useful at most programming stages than precision. In the limit, as sampling approaches infinity, RIF becomes QIF. RIF also maps information in a way that is

easier for the non-specialist to interpret. One does not already need to know what to look for, the scalability of the approach means that everything can be observed and understood in context.

3.3.2 *FlowForward*

An example IT measure is *mutual information* (MI). The MI between two random variables, X and Y, is given by

$$I(X;Y) = \sum_{x \in X, y \in Y} p(x,y) \log_2 \left(\frac{p(x,y)}{p(x)p(y)} \right) \tag{3.1}$$

This tells us how much information is shared between two random variables, X and Y. If we know both X and Y perfectly, then it is easy to calculate MI. In all of our examples to follow, X is the distribution of inputs to a function, and Y is the output of that function. As an example, a function might take a single integer argument: in actual program execution, this integer might only occur within some range, and some integers might occur very frequently, some very rarely. In practice, we rarely know X and Y perfectly and have to approximate them through sampling. If we test a function on just a handful of inputs, the MI picture is less accurate than if we test with many inputs. Ideally, we also have to test with the right inputs: those that we actually see during execution.

For example, let us define a leak as any unwanted movement of information, and *quantify* the size of that movement using MI. A leak of 1 bit from a boolean valued function is a leak of *all* the information in that function, whereas a leak of 3 bits from a 64 bit integer valued function is probably rather little. To the specialist this is all rather obvious, but most software engineers are not specialists, and it is software engineers who write the code and consider code security. To rank these correctly, it is more useful to know how much of the input is reflected in the output. With these requirements in mind, we introduce a new information

theoretic measure which we call *FlowForward*. *FlowForward* is the proportion of information contained in X that is reflected in Y (Figure 3.3).

Definition 3.3.1 (*FlowForward*)**.** Given 2 random variables, X and Y, and a *criticality weighting* δ, the *FlowForward* from X, $F(X,Y,\delta)$ is given by

$$F(X,Y,\delta) = \delta\left(1 - \frac{H(X) - I(X;Y)}{H(X)}\right) \tag{3.2}$$

This does not make assumptions about the nature (whether total or pure) of the function, *i.e.* $H(Y)$ can be greater than $H(X)$. This might be the case if a program function has access to state that is not captured in its input parameters, for example. $H(X) - I(X;Y)$ is just the conditional entropy of X given Y, so $F(X,Y,\delta)$ can be rewritten $\delta\left(1 - \frac{H(X|Y)}{H(X)}\right)$. The *criticality weighting*, δ, allows a user to include a security policy or some other form of model into the map. We assume δ is uniformly 1 in our examples, but a simple confidentiality policy could be modelled by setting $\delta > 1.0$ for any nominated function. $F(X,Y,\delta)$ is guaranteed to be a value between 0.0 and 1.0 when δ is 1.0. Given the fact that $I(X;Y) \leq H(X)$ then we have, in the case of no mutual information, $1 - (H(X) - 0/H(X)) = 0.0$, as expected, and in the case of total flow $1 - (H(X) - H(X)/H(X)) = 1.0$. This allows us to compare different functions and methods which are different in their supports.

The presence of δ allows us to scale *FlowForward* based on assumptions of importance. For example, a simple security policy might say that all user input should be regarded as suspicious. By defining $\delta > 1$ for these functions, they will have a higher *FlowForward*, which will result in greater prominence in the ICM. If, even after applying δ, the *FlowForward* of a function is low, it is likely that the information introduced by a user is having limited effect on the computation. Knowing the exact values of X and Y is, as we have seen, very difficult. In practice

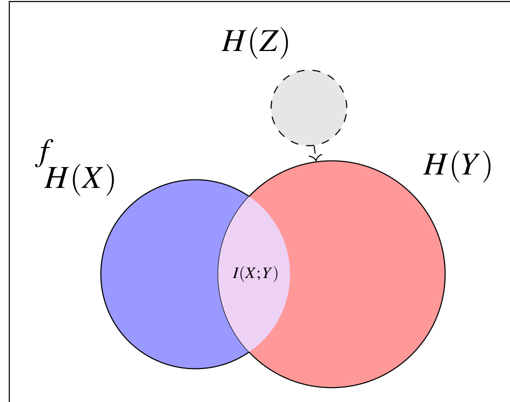

Figure 3.3: *FlowForward* is 1 - the area in blue, normalised by $H(X)$. This allows us to think of *FlowForward* as the proportion of information in $H(X)$ that reaches $H(Y)$. Let f be a function with input X and output Y: if $H(X) = 2.3$ and $H(Y) = 4.0$ and $I(X;Y) = 1.3$, then the *FlowForward* $R(X,Y,\delta)$ (assuming $\delta = 1.0$) is $1 - \frac{2.3-1.3}{2.3} = 0.56$. The normalised deterministic squeeziness ($\frac{H(X)-H(Y)}{H(X)}$) is $\frac{2.3-3.5}{2.3} = -0.74$, indicating the there must be an additional source of information inside the body of f which is not coming from X. This addition source of information is represented by $H(Z)$, state accessed inside the body of f.

we have $\hat{F}(X,Y,\delta) = \delta\left(1 - \frac{\hat{H}(X)-\hat{I}(X;Y)}{\hat{H}(X)}\right)$; the estimated value of *FlowForward* for random variables X and Y. This estimate may fall below 0, or rise above 1 due to noise in the estimated values of $\hat{H}(X)$ and $\hat{H}(Y)$.

3.3.3 Interior Information

FlowForward is excellent for telling us about how much information passes through the function but it does not tell the whole story. If observations are taken at the granularity of the function, then some important information may be missed. If a function calls on global state within its body, *FlowForward* cannot directly

detect this. To solve this problem, we follow up a suggestion from Clark *et al.* [57] and we repurpose another IT measure, *squeeziness* [53], specifically *normalised (deterministic) squeeziness* (*NSq*), Equation (3.3) [54]. It measures how much a total, deterministic function compresses information.

It finds application in work on fault masking and failed error propagation (Section 3.7). Normalised squeeziness is defined as $H(I|O)/H(I)$, a Shannon measure and always positive. For deterministic functions we can use the simpler

$$SQ_N(f) = \frac{H(I) - H(O)}{H(I)} \tag{3.3}$$

as f being deterministic means $H(O) \leq H(I)$ and $SQ_N(f)$ yields a positive real number. If f is not deterministic then we are no longer, technically, measuring squeeziness, but performing a relative comparison between input and output entropy. It is no longer a Shannon measure. It does, however, allow us to flag functions which feature some non-deterministic or stateful components. A negative result is due either to hidden state or some source of non-determinancy, as in Figure 3.3. By outputting both the *FlowForward*, showing how much of the input carries through to the output, and the *NSq*, if negative, we can build a detailed map of how information moves inside the SUT's call graph (Figure 3.7).

3.3.4 Data and Entropy Estimators

To improve estimate efficiency, RIF uses state-of-the-art entropy estimators. As previously discussed in Section 3.2), calculating entropy via Monte Carlo methods is an expensive problem. The simplest entropy estimator is the *Maximum Likelihood Estimator*, also known as the *Plugin* estimator. This takes the count data from observations and slots the resulting histogram into the entropy formula. Such an approach, while simple, on average, underestimates entropy [22]. This realisation has led to a large amount of research into how best to estimate entropy from count

data.

Some well-known estimator improvements are Grassberger [118] and those in Planinski [151]. Essentially, they all add a bias to account for undersampling and differ in how they calculate that bias. Several estimators specifically for MI exist, such as those proposed by Zeng *et al.* [225], and Hernández *et al.* [105]. The most flexible, in that it handles mixed discrete and continuous data, is the *knn*-based estimator of Gao *et al.* [88]. While we tried this algorithm, it is unclear how best to automatically set the value for k. Moreover, *FlowForward* requires an entropy estimate, not just an MI estimate.

The idea behind Bayesian entropy estimators is simple: specify a model relating observations to the unknown quantity, then compute the posterior given the observations. In particular, we use the NSB (Nemenman-Schafee-Bialek) algorithm [144]. This algorithm allows entropy estimation even when the number of samples is much smaller than the size of the alphabet, where the alphabet is the number of classes with a non-zero probability. Given the extreme difficulty of knowing the alphabet size for any given datum, this flexibility is extremely important for our approach. When we have custom objects for our random variable, we really do not have a ready means to assess the cardinality of that object. When such information is not present or known (as will most often be the case), we use a variant on the NSB algorithm for random variables with large cardinalities [142]. In effect, this simulates a strongly undersampled regime for countably infinite spaces.

Continuous Data: We use the "3H" approach to calculate mutual information: $I(X;Y) = H(X) + H(Y) - H(X,Y)$, where $H(X,Y)$ is the entropy of the joint distribution. While this has some limitations in higher-dimensional continuous spaces, it is sufficient for our purposes. MI estimators for such spaces exist [88], but it is

not clear how to apply them to entropy estimation. Their use is, so far, largely confined to machine learning applications. Continuous data is traditionally handled via discretisation, as the entropy of a continuous distribution is infinite.

3.3.5 Noise Reduction via Clustering

In any SUT there might be several hundred active observation points. It stretches credulity to assume that the ranking of these points' *FlowForward* will remain stable over time. Even the best entropy estimators will fluctuate when a variable's distribution changes. We also have the problem of "circling" functions: those whose estimates circle each other again and again without reaching a fixpoint. Such functions will always perturb rank correlations, even though the essence of the ranking approach asserts there is no useful difference between an estimate of 9.05 and 9.1.

Our solution is to group functions with "similar" *FlowForward* and assess rank similarity over the groups. The problem then is to define "similar". Initially, we experimented with various clustering algorithms. These operate on the assumption that this is an underlying grouping to the data, and often require an initial guess as to how many groups there are. We quickly realised that we could not make such assumptions. There is no reason to assume clear groupings in the information flow properties of functions, and we had no idea how many groups there might be. We resolved this problem through much simpler mathematics: rounding. We group similar functions by rounding their estimates to the nearest $.x$. We control how many groups there are by careful specification of x. This solves the problems of "circling" functions.

The only subtlety we permit ourselves in this simple system is we always round in the same direction: up. We note that rounding induces a Galois Connection [217]. Let the set of raw *FlowForward* scores be partially ordered $(\mathbb{R}, \leq)$.

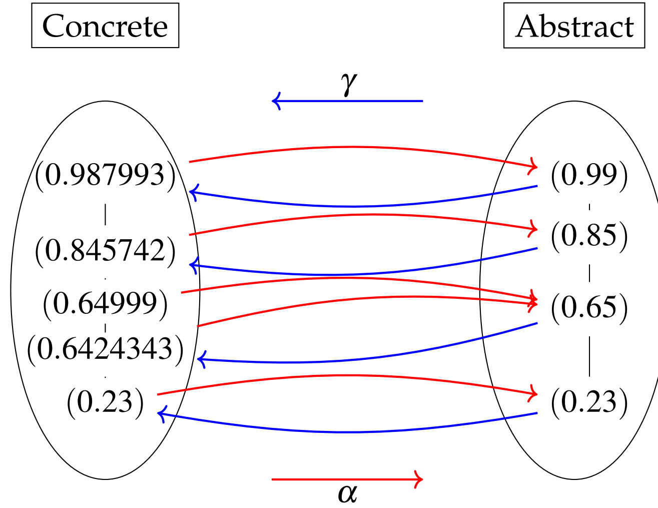

Figure 3.4: Lattice A contains the raw RIF scores, partially ordered by value, derived from testing the program, while lattice B is the abstract lattice of those scores with the same partial order. The red arrow is the abstract function α. Here α is interpreted simply as round up to the nearest $.xx$. The concretisation function γ (in blue) maps the abstract value to the nearest value below it, thus satisfying the requirements for a Galois Connection, $\forall a \in A, b \in B : \alpha(a) \leq b \textit{ iff } a \leq \gamma(b)$. We compare rankings over the abstract lattice.

Further, let $\alpha_n : \mathbb{R} \to \mathbb{R}$, so that $\alpha_n(x) = x'$, where x' is x rounded to n decimal places. α_n is a monotone map $x_1 \leq x_2 \to \alpha_n(x_1) \leq \alpha_n(x_2)$, extensive $x \leq \alpha_n(x)$, and idempotent $\alpha_n \circ \alpha_n = \alpha_n$, so it is a closure map. We use it as our abstraction map but define γ as the largest data point less than the abstraction This defines a Galois Connection in the ordered set of real numbers.

3.4 RIFFLER Implementation

RIFFLER is written in Python. It consists of two parts: an annotator and a *Flow-Forward* estimator. It uses decorators to capture function call information. These

decorators are the only language specific part. There is no absolute need to use decorators, but they make the process of applying annotations much easier. In return for the ease of application, the granularity of automatic observation is at the level of function. One can, of course, write manual observation points at any level of granularity. We provide a decorator for Python, and an decorator for Rust. The annotator automatically applies the decorator annotations to Python code but, for Rust, annotations are currently manual.

The second component of RIFFLER performs the information theoretic calculations. This uses a Bayesian entropy estimator; we discuss why we chose this and how it works in Section 3.3.4. Observations are stored in a NoSQL database, by default, MongoDB.

3.4.1 Capturing Observations

RIFFLER uses the *dill* serialisation library [134]. We record the input arguments to the function and serialise them using *dill*. For reasons of efficiency (of both comparison and memory use), rather than dumping the bytestring output directly to the Mongo database, we store the 128 bit *murmur3* hash [218]. We use a constant seed to allow us to compare different runs of the program. We assume the input to be trusted. Hashing preserves identity only: this is not a problem for information theory, as a count of identities is all that is required. We do not need to know *what* is being observed.

RIFFLER converts the list of input hashes to a *csv* string before storing them in its Mongo database. We use the same procedure for the output of the wrapped function. For functions that mutate the input values and do not return a value of interest, our solution compares the hashed snapshot of inputs *before* calling the wrapped function, with the hashed snapshot of inputs *after* calling the wrapped function.

We record the occurrence count for each unique input/output pair of a function. From a purely practical point of view, this saves space in the database. As a happy byproduct, this also permits using the count as a means of assessing the input/output diversity of our testing regime. Given that a uniform distribution is the most informative, we can easily compare the RIF of a function *ignoring* the occurrence count and *with* the occurrence count. Similar values means that the multiplicity of input/output pairs has had little effect on the RIF, so we have a distribution approaching uniform.

Certain functions, such as `__init__`, tend to be called infrequently while fuzzing. They are important, so we do wish to include them in the callgraph, but we also cannot give a *FlowForward* estimate with a large degree of confidence. For functions called fewer than 5 times, we use the Grassberger estimator (Section 3.3.4), as NSB does not function under these circumstances. The output of RIFFLER makes clear where there was insufficient data (Section 3.6).

3.4.2 RIFFLER Discussion

While we have attempted to make RIFFLER as robust and usable as we can, there are still a few areas where manual intervention is required or limits to the testing regime are exposed. We discuss these, and our solutions, below.

RIFFLER comes with a Python3 annotation script for adding and removing RIF annotations. This annotation is a Python decorator that is added before each `def` in the source code. We ignore certain files in a project by adding them to a global *ignore* list. This includes, by default, such files as *setup.py*, which is an artefact of Python *setuptools* and not part of the source code proper.

RIFFLER requires a test harness, provided as a command line variable. This file needs to be a self-contained Python script. When testing library code, only those functions callable through the testing harness are observed. It can easily be

the case that a large number of RIF annotations are added via the annotation tool that will never be observed from the given entry point. This indicates that a better test harness is required.

While the Python RIF decorator is quite robust, it does have limitations. For instance, some functions contain code that *dill* does not serialise, due to infinite recursion. This infinite recursion leads to a stack overflow and causes the Python instance to crash. It is not possible to recover gracefully from this crash, nor to catch it before the damage is done. Such annotations need to be removed.

At the moment, RIFFLER annotations are immutable during a testing run. This means that, even if a sufficient number of samples has been gathered to have a good confidence that the *FlowForward* is correct, RIFFLER takes a testing performance hit from the decorator. Future work will edit the AST on the fly to remove exhausted annotations to improve testing efficiency. RIFFLER's annotation imposes a runtime cost. As it is a testability transformation [101], production code does not pay it.

Continuous Data: Our entropy estimator only handles discrete data. A continuous variable has infinite entropy; the usual method to tackle this is to discretise the data. Our method of hashing every variable to a 128 bit integer and using an asymptotic entropy estimator maps the uncountably infinite space of the reals to a countably infinite space. This transformation is therefore lossy. While this technique is in line with standard practice, future iterations of RIFFLER should allow bucket sizes to be set by the user in the event that the information is known, or bucket size is important.

Confidence: RIFFLER does not provide statistical guarantees on its map by default. Other tools provide such guarantees, at the expense of more upfront knowledge and expertise. For the user who requires such confidence, RIFFLER

can use the standard deviation of the entropy estimate as a stopping criterion. The NSB algorithm, being Bayesian, returns a normal distribution, specifically its mean and standard deviation. It is a simple task to require Riffler to keep testing until the standard deviation is sufficiently low.

False Constant Functions: Information theory does not care what is observed, just how often it is observed. Take for example the function `parse(*args, **kwargs)`: it might be expected to have a high RIF, as parsers usually convert information instead of throwing it away. The `parse` function, however, might return a pointer to an object <obj.Parse.parser at 0x343f343e> instead of a detailed view of the input stream and partial AST. Observing only this memory address leads to the *false constant problem*. All sorts of changes may have occurred internally to this object, but the observation point value itself never changes. As a result, `parse` appears to be a constant function. Constant functions have a very low MI, and therefore very low RIF.

Is the reported RIF for false constant functions correct? Yes, in a purely information theoretic sense. In a practical, useful, sense, we want to minimise the presence of these false constant functions. To expose the output of such functions, Riffler attempts a full serialisation of the object. We use dill to deference all internal pointers. When full serialisation is impossible, due to recursion limits, we resort to a shallow serialisation of the top level object members only. Riffler records the occurrence of shallow serialisation. If even this fails, Riffler resorts to *very shallow* and simply directly hashes the returned value.

Fuzzing: We use fuzzing for ease of testing. Fuzzing requires only one entry point. It should be possible to generalise a Python test suite to produce random input values, but this is not something that Pytest or similar tools does automatically. This introduces the problem of creating more "junk" inputs, *i.e.*

inputs that are not likely to ever be seen in real deployment but which do not cause an error; a problem from which fuzzing suffers less. This could potentially distort the input distribution to the point that then entropy estimate is misleading.

Deployment: RIFFLER is intended for deployment into a project's testing regime. RIFFLER can automatically annotate individual files or entire directories for testing and automatically removed them for release. This is the only manual step for most projects. As discussed above, testers will need to remove some annotations due to recursion difficulties. Future versions of RIFFLER will aim to handle this problem more elegantly by automatically detecting such functions. RIFFLER is independent of testing regime. As long as a function is called a sufficient number of times with diverse inputs (*i.e.* enough that the NSB algorithm can return a result), then a contour map can be produced. In general, the better the testing regime, the better the map.

RIFFLER is a prototype tool for RIF measurement. It requires pure PYTHON programs for decorator application. This acts as a limit on the size of program testable: not because of an inherent limitation in the approach, but rather limitations in PYTHON. Large PYTHON projects, such as *numpy* and *pandas* are frequently written in C, with a PYTHON interface. We cannot yet produce maps for these large and interesting projects, as C lack decorators. Further engineering work is required to solve the problem of capturing I/O pairs automatically for C functions.

3.5 Evaluation

The ICM that RIFFLER produces is only meaningful if the ranking is meaningful. Given an unlimited testing budget, RIF approaches accurate QIF. In the limit, we know that RIF produces useful results. In the real world, unlimited testing budgets

are unlikely. The question then becomes what is the least amount of work that we can do to obtain reasonable results. We answer this question experimentally.

Experiments: For scalability and utility, we show that estimating *FlowForward* produces meaningful results *w.r.t* the ground truth of information flow within the SUT. The chief difficulty is establishing the ground truth. As we have seen, obtaining a bounded MI estimate with a 95% confidence interval, for even one random variable, can be prohibitively expensive (Section 3.2). We tested each program for 24 hours and took the final ordering as the ground truth. We use the Kendall rank correlation coefficient [216], τ, to measure the similarity of the orderings at each hour compared to the ground truth. We exclude functions that are chronically under-sampled. Kendall τ requires that both orderings are the same length. As fuzzing progresses, it may find more functions, so the orderings will be different lengths. To calculate interim orderings, we truncate the ground truth to include only those functions in the interim ordering.

Experiments were conducted on Microsoft Azure cloud computing. We used RIFFLER to add annotations. A total of 10 annotations were removed across 3 programs due to serialisation limitations. We tested each program for 24 hours, either with a seed corpus, or a HYPOTHESIS test suite. We tested each program 10 times, and present the average of the results in Table 3.1. In one case, *ruamel*, a slow unit report caused fuzzing to terminate earlier than 24 hours. RIFFLER attempts to automatically restart until the required time has passed. In our experiments, we used the *atheris* fuzzer from Google [96]. This is a Python front-end for the LIBFUZZer project. We used the ENTROPIC scheduler to improve code coverage.

Corpus: Our corpus consists of the programs listed in Table 3.1. As no existing corpus dedicated to fuzzing Python is known to us, we built our own. Two programs in the corpus are derived from the sample fuzzers included in the

ATHERIS GitHub repository. We drew the other programs uniformly at random from a search of projects on GitHub. First, we filtered to find only those projects which had existing Hypothesis test suites. Next, we removed projects which were not largely in Python. The prototype RIFFLER tool cannot yet add annotations to non-Python code. We filtered this list to remove tutorials: we only considered library code and executables for inclusion. While this last stage is not essential, we wished to explore real-world code first and foremost. Full details of this search procedure are in Section 3.9. We chose 8 programs, with a total of 12 distinct testing entry points, as a large enough sample from which to draw conclusions.

Many fuzzing papers, in particular, use fewer programs. Our corpus suffers the usual threats to external validity. We created a corpus to match our computing resources and RIFFLER's current limitations, but we have no reason to believe these concerns bias our corpus selection in RIF's favour.

RIF Performance: If ranking is to scale, it needs to approximate the ground truth before exhaustive testing. We fuzzed a total of 11 entry points drawn from 8 different Python programs. As many of these projects are libraries rather than executables, different testing entry points had to be prepared. The results are presented in Table 3.1. Functions tracked is the total of all functions called during testing, with total calls being the complete amount of work done over the 24 hours. Under-sampled is the number of functions for which there are insufficient samples, even after 24 hours, whereas Shallow covers both shallow and very shallow serialisations. We tracked a total of 768 different functions over the course of the experiments. Rankings over individual runs were quite stable; this is likely due to the use of the same seed corpora or test cases for fuzzing.

From Table 3.1, we can conclude that for most programs, under-sampling and shallow functions were not a problem. The number of under-sampled functions in

`idna` is rather startling. Only 17 were sampled sufficiently to obtain good quality estimates. The shallow sampling of `ruamel` is the result of regular expressions in the code; these are difficult to serialise; though can be explicitly ignored. The virtue of different entry points while fuzzing is displayed by `jamespath`: fuzzing the search API covered 75 functions, whereas fuzzing through the main entry point only covered 21. Shallow functions (and, by extension, false constant functions) were not problematic in our corpus, though they did account for 22% of all tested functions in Fernet. Figure 3.6 shows RIF's performance. We found that, on average, ranking relative to ground truth started at 92%. While testing for even an hour produces a good quality map, testing for longer produces a bigger map, as new code is found by the fuzzer.

As previously discussed, obtaining a ground truth with another tool, such as LEAKWATCH, is computationally unfeasible. However, it is possible to calculate the *minimum* number of samples that LEAKWATCH requires given a particular input/output space. This is four times to product of unique inputs and outputs per observation. We calculate this and compare it against the actual number of observations. As we have used fuzzing as a test driver, some functions are over-sampled with respect to LEAKWATCH requirements. These functions include `__init__` functions and similar object creation activities. We exclude these functions from our analysis, as the over-sampling is a byproduct of the testing regime, rather than a necessity of ranking.

Over our corpus, we find that ranking requires only 40% of the lower bound number of samples required by more precise estimation. This is a minimum 60% saving on number of executions, as shown in Figure 3.5. A *z score* analysis of the results does not reveal any outliers until setting the score at $1 \leq z \leq 2$.

Threats To Validity: We address the external threat our corpus poses above.

Program	Functions Tracked	Total Calls	Under-sampled	Shallow
ruamel	161	373824	0	41
chardet	120	314562	0	0
ecdsa	57	363937	8	0
idna	95	404682	78	0
hazmat*	53	398978	0	15
fernet*	82	363572	17	18
dateutil	20	366603	6	1
james_main**	21	381898	5	4
james_search**	75	394276	6	7
james_api**	63	378872	7	7
ssh-decorator	21	387330	0	1

Table 3.1: RIFFLER performance on Python programs using atheris. The total number of functions ranked over the corpus is 768. Shallow readings are only problematic in a few cases. Large under-sampling is, except in the case of `idna`. Entries marked * are different algorithms in the Python Cryptography library, whereas those marked ** are different parts of the API for the `jamespath` library.

RIFFLER has two limitations that further threaten its external validity: slow unit reports and non-observable functions. Serialising objects takes time. This additional cost sometimes leads LIBFUZZer to spuriously report *slow unit*. Further work is required to disentangle the overhead of RIFFLER's decorations from non-functional characteristics of a decorated function. A small number of functions could not be easily serialised and were dropped. This serialisation limitation is

Figure 3.5: Rankings require far fewer samples than more exact estimation methods. RIF needs 60% fewer observations than the minimum required by LEAKWATCH for a particular set of input/output observations.

as a result of *dill* not being able to serialise some parts of the Python language, especially regular expressions. We have no reason to suspect that dropping them affected RIFFLER's performance. Even if we are wrong, and filtering introduces systemic bias, the proportion is so small any resulting bias would also be small.

3.6 Case Studies

We present several RIFFLER case studies. We have already seen, in Section 3.2, that malware can be discovered by RIF mapping. We look at several popular Python projects to discover what an ICM can tell us about them. Each ICM presented was extracted from the larger callgraph by looking for large change in *FlowForward* between functions. This helps to reduce the complexity inevitable in the graph visualisation of software. The colour schemes for the ICM were chosen from a

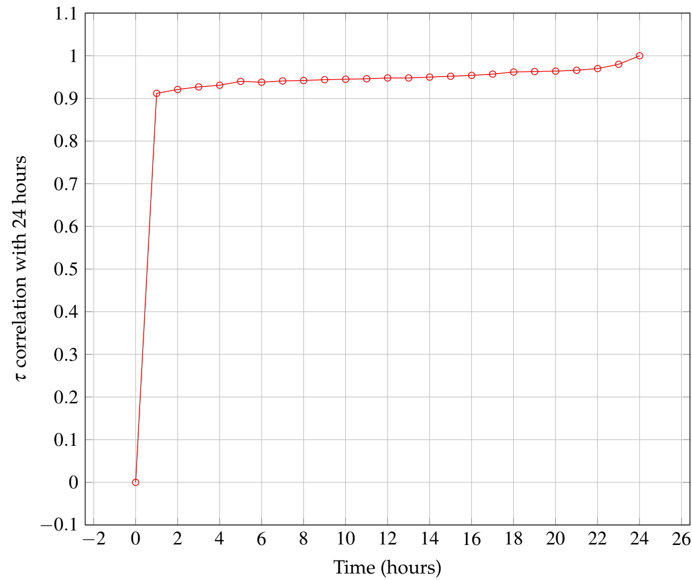

Figure 3.6: The more testing resources spent on RIFFLER, the larger the map: as the fuzzer finds more functions, the map grows. The accuracy of the map also improves, but already starts at 92%. Even a small amount of testing produces a good approximation of the result obtained at 24 hours.

cool heatmap colour scheme. This colour scheme can be changed in our prototype RIFFLER tool.

Cryptography: The Python Cryptography module provides cryptographic recipes and primitives. This module is mostly written in Python, with a small amount of code in Rust. We show a small extract from its ICM in Figure 3.7. While most of the ICM has low RIF, the interior information of `finalize` jumps out. It draws our attention to the presence of an information source not present in this slice of the ICM. Where does this information come from, and can it be trusted? Closer inspection of the function reveals that it is receiving a lot of information via calls to C through the foreign function interface (ffi). There is a great deal of trust being placed in these foreign calls. Prudence suggests additional testing and

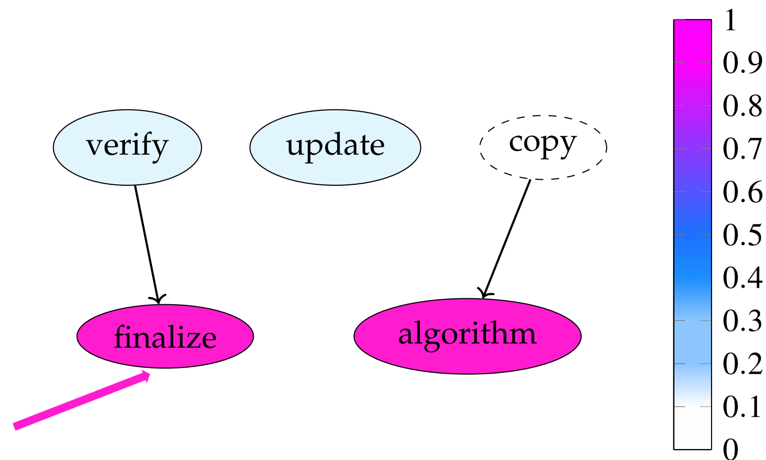

Figure 3.7: An extract from the `hmac` module ICM in Python's cryptography library. The `copy` node is dashed because too few samples were taken to make a good estimate. The fat arrow going into `finalize` is negative normalised squeeziness (interior information) and indicates an information source other than the function's input. This information source is large. Is it legitimate?

annotations here might forestall or reveal leaks in the future.

ECDSA: The Python ecdsa library [201] is a pure Python implementation of the Elliptic Curve Digital Signature Algorithm. It allows the user to quickly create key pairs for both signing and verifying. We examine a part of the ICM in Figure 3.8. The first obvious feature is the relative lack of connectivity between the nodes of the ICM. The low RIF of most of the functions appears linked to the low connectivity. These functions are setting up state rather than returning values of interest.

Of more interest are `sign` and verifies. The `verifies` function has a RIF of 0.12 and an interior information of 0.88. Examining the code, one sees that it returns a boolean value: it verifies that the signature is a valid signature of the provided hash. A lower RIF might be expected here, given that most of the

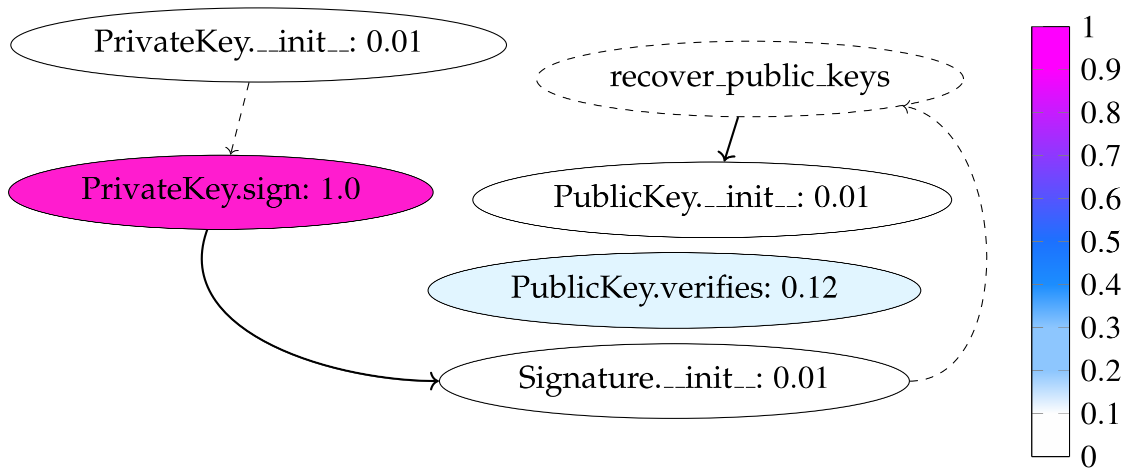

Figure 3.8: Extract from the ICM of ecdsa. Information groups form within the map, those with low RIF being associated with initialisation. Note that the verifies function has similar RIF to the verify function in Figure 3.7.

information provided is not passed on. Whether it constitutes a leak or not is project dependent, but the user now knows that over 10% of the input entropy flows through this function.

The function `sign` is an interesting example where a criticality weighting, $\delta < 1.0$, might be applied. It preserves, but converts, the information given to it. Its high flow is expected, but it is not a leak, as the information is encrypted. Now that we are aware of it, we can give it less weight in the visualisation during subsequent testing.

IDNA: The Python library, idna [64], is an implementation of the Internationalized Domain Names in Applications protocol as specified in RFC 5891 [116]. Figure 3.9 shows an extract from its ICM that RIFFLER produces for it. This extract makes three actionable suggestions:

1. a name refactoring;
2. error condition testing is inadequate; and
3. a potential mismatch between function names — `encode` and `decode` — and their flows.

One can immediately see that the ICM splits into two groups. One group is the low RIF group, containing all the functions with the prefix `check_`. The function `valid_string_length` has the same RIF as the `check_` group, but a different naming convention. Looking at the ICM, one might choose to rename `valid_string_length` to ease code navigation and aid code completion tools.

Another feature that leaps out is that `IDNAError` was not called during 24 hours of fuzz testing. This speaks well for the robustness of our code, and badly for the robustness of our testing regime. It is clear that both `encode` and `decode` preserve a great deal of their information inputs, and also dump directly into `IDNAError`. Seeing this, a developer should consider writing tests that make sure to exercise this path most of all. While tests should also be written to exercise the path from `check_hyphen` to `IDNAError`, this is of lower priority. Why? Because the information flow from `check_hyphen` is very low, whereas from `decode` it is very high.

It seems both `encode` and `decode` compress, while their names suggest lossless encoding. They do not have the same RIF: moreover, `encode` has a normalised deterministic squeeziness of 0.87 and `decode` 0.05. This low information compression of `decode` is likely to be a statistical fluke, brought about by error margins in the entropy estimator. But it is certainly worth investigating to ensure it is not a programming error. Closer inspection of the code suggests that the information

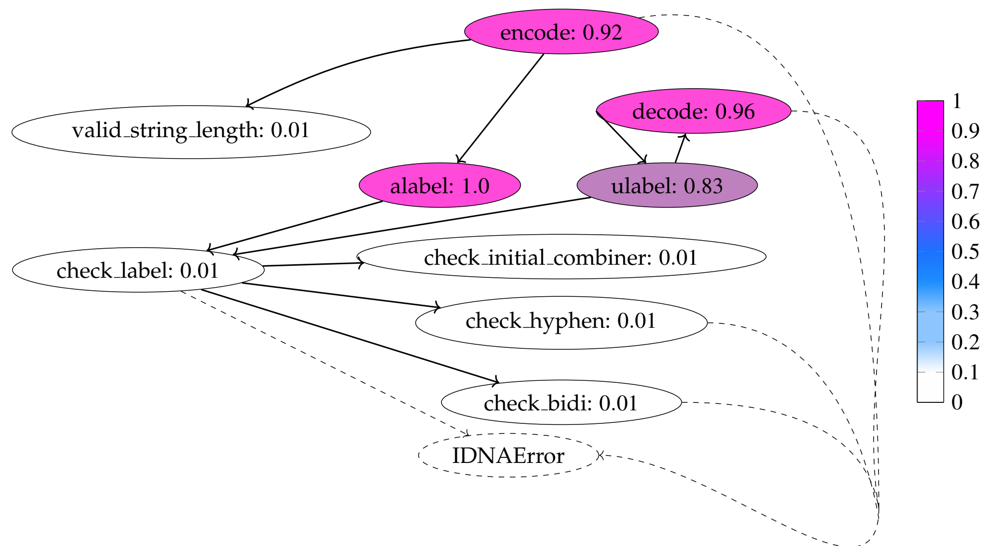

Figure 3.9: An extract from the ICM of `core.py` from idna. One can immediately see that one can group them by information characteristics. All of the functions called `check` have the same RIF and are connected in the ICM. Both `encode` and `decode` have similar, but not identical, RIF. Both functions are slightly lossy, `encode` more than `decode`. The dashed lines and dashed node on `IDNAError` mean that this function was not called during testing.

loss for both functions is due to the presence of default parameters. It is likely that these were not altered during testing and so their information was "ignored". This suggests for improving our testing regime.

3.7 Related Work

As far as we know, we are the first to discover information policies via mapping within a computer program. Mu [138] proposes quantified program dependence graphs; while these have a superficial similarity to our ICMs, they do not map

information flow through systems, and only function for a simple *while* language. Our ICMs map information flow in real software. We achieve this scalable mapping by grouping and ranking flows by magnitude. We are the first to propose this method and provide experimental data that such a method works and has practical value. Existing tools, as discussed below, assume that the user knows the policy, and quantifies deviation from policy for some variables. Such an approach is undoubtedly useful when considering protecting secret values, but it does little to help us when security concerns are underspecified. IT has recently been shown to have applications outside of program security (Section 3.7.1). Again, the main barrier to acceptance is the problem of obtaining sufficient observations to make estimates. We briefly examine work in this area before moving on to the traditional home ground of QIF, information security.

3.7.1 Information Theory in SE

While information theory for SE is still in its infancy, it has already shown its utility. We focus on two different applications, one of which has impacted the state-of-the-art in fuzzing, and another which improves test suite health.

ENTROPIC [28] is an extension to LIBFUZZer [125] which uses information theory. By prioritising the seed with the largest entropy for continued fuzzing, ENTROPIC is also using ranking: it implicitly recognises that the precise amount of entropy is not important, just how much it has in relation to other variables.

Failed error propagation (FEP) is one known cause of *coincidental correctness*: the situation where a program contains faults, but still produces the expected output. Androutsopoulos *et al.* [10] define FEP with information theoretic setting based on conditional entropy. The paper demonstrates that knowing relative magnitude of conditional entropy is sufficient to detect the presence of FEP. RIF shows that relative ranking is stable and efficient and our tool, RIFFLER, could be

used to provide this information. This would improve both code and test suite quality.

3.7.2 Information Theory in Computer Security.

RIFFLER is a new direction in the security sphere, yet is complementary to existing information theory tools to program security. There are many tools attempting to solve the problem of places bounds on information leaks within a program; we examine the most relevant and discuss how they fit with RIFFLER.

The nearest research to RIFFLER's approach is *dynamic information flow analysis* (DIFA) [130]. This is both a profiling and enforcement mechanism. It uses a program slicing analysis which focuses on precision. It requires a separate testing phase and is also used during deployment. DIFA does not use RIF to scale its analysis. Moreover, RIFFLER is a testing transformation with an associated flow analysis mechanism that fits into a normal testing campaign. With RIFFLER, unlike with DIFA, the user has direct access to observation points; changes are source level. A user is free to add or remove annotations, or write tests that explicitly target a part of the code. This makes RIFFLER a highly flexible generalisation of DIFA. Our method of storing the point-to-point variable observations allows a wealth of other analyses to be performed in addition to those presented by Podgurski *et al.* [130].

DIFA uses *StrengthFlow* [131], performing a similar task to *FlowForward*. *StrengthFlow* is defined as $\text{StrengthFlow}(X,Y) = H(X) - H(X|Y)$. *FlowForward* has the advantage of normalisation, and is more easily interpreted as the percentage of information in X that occurs in Y. The normalisation allows us to perform ranking based on the proportion of the input which is shared with the output. No previous DIFA work suggests using ranking to simplify the resultant policy picture as we do here. *StrengthFlow* lacks the *criticality weighting*, δ, which allows

us to model expert knowledge within our policy map.

Given the number of papers which consider the question of precise, or bounding, QIF, it is impossible to cover everything here. Instead, we discuss a few approaches which are most relevant to RIFFLER. We believe that each method is usable in conjunction with RIFFLER, after the preparation of an ICM.

F-BLEAU [46] is a black box leakage estimation tool that uses machine learning techniques to provide an entropy estimate based on samples of input/output behaviour. This could be used as a replacement for the NSB entropy estimator: this requires further investigation before drawing conclusions about which is more effective and cost efficient. Other machine learning approaches, such as that of Romanelli *et al.* [160] might serve a similar function.

The LEAKWATCH tool for Java [49] uses a flexible point-to-point leakage model which is similar to RIFFLER. It allows the insertion of arbitrary observations points into source code. One can introduce these observations at the statement level. RIFFLER, by default, works at the function level but there is not reason, other than user effort, that its observations cannot work at another level of granularity. The advantage of workings at the function level is we do not need to make decisions about what to observe and where. Such decisions imply policy knowledge and understanding, which we believe is unlikely in large projects. RIFFLER provides that policy knowledge, but can also provide the details of LEAKWATCH. RIFFLER maintains the a separate estimate for each function observed and ranks them with respect to each other: LEAKWATCH returns only *one* value for the entire program and cannot be used as a mapping tool. One has to assume that annotations are correct and in the correct places. RIFFLER does not make such an assumption.

Biondi *et al.* [26] present an experimental comparison of different leakage

estimators. These leakage estimators mostly focus on one or two variables, and assume the existence of a security policy. Moreover, all of the tools make the assumption that one knows where to look for problems in the code. RIFFLER identifies problems when the policy is unknown or poorly understood and can be used in conjunction with any of the other approaches. This is especially useful when considering tools based on symbolic execution. RIFFLER can identify those parts of the code that need more careful examination. Mapping an entire system using methods based on symbolic execution would be extremely expensive, whereas RIFFLER is lightweight.

Backes *et al.* [20] present a method for automatically detecting and quantifying information leaks from a program. This provides a detailed IT analysis via logical relations and symbolic execution. Their tool requires a model checker, with all its attendant limitations. RIFFLER has no such requirements and only requires observations. RIFFLER is a much more lightweight approach which is also much more transparent to the user.

McCamant *et al.* [132, 133] use network flow capacity to model information flow security in C-language programs. Their tool requires multiple program runs to provide a bound on leakage within the SUT. This is an additional testing regime, whereas RIFFLER integrates into normal testing. As they say in their discussion for FLOWCHECK, it is envisaged as an auditing tool for a security policy, rather than as a means for creating or discovering a policy. RIFFLER is both, providing a map of information flow without prejudice to policy, and the ability to "drill down" into more detailed flow exploration.

Finally, we highlight a distinction between our dynamic flow analysis and unrelated work with a similar name. The term *dynamic information flow analysis* is, unfortunately, overloaded. It can refer to the DIFA work, as discussed above, but it

can also refer to a body of work which is not directly concerned with information theory and quantification. Austin *et al.* [16, 18] use this name to describe various methods for ensuring non-interference [95] at runtime. This relies on the presence of data labelled *w.r.t* some security policy, and makes no attempt to quantify the size of any potential leaks, tracking label interactions instead.

3.8 Conclusion

We have presented RIF, a scalable abstraction of QIF for software information mapping. We show that RIF allows us to build an abstract picture of information flow within a program using far fewer samples than previous methods. We have also presented *FlowForward*, a new information theoretic measure for mapping information flows through the callgraph of a program. In addition, we have presented a new use for normalised squeeziness, showing that it allows us to identify hidden sources of information within the body of function during greybox testing.

We introduce a tool, RIFFLER, which creates a RIF image of the SUT. RIFFLER is easy to use, automatic, and incorporates seamlessly into software testing. We anticipate that RIF will bring information theory to the non-specialist developer, allowing them to explore and define security policy for software in an interactive and evolutionary manner.

3.9 Appendix: Creating a corpus of Python Projects

The corpus was created from open source projects hosted on Github `https://www.github.com`. The Github API allows programmatic access to public repositories: it requires the creation of a access token. Alternatively, the web interface provides the same search functionality, but data presentation is more difficult to parse.

The main search query was `''from hypothesis''language:python`. The Hypothesis library is used by more that 4% of Python users, according to the Python Developers Survey 2020 `https://www.jetbrains.com/lp/python-developers-survey-2020/`. As Hypothesis is almost always used as a qualified import, the search term looks for the most common form of this. The search term will miss repositories that import Hypothesis directly without qualification. Projects were then selected, uniformly at random, from the search results and then subjected to the following manual analysis:

- project must be on PyPi `https://pypi.org`. This was to remove tutorials, lab work and homework repositories.
- the project must be largely in Python. This information is available on the Github page of each project. "Largely" was taken to be 85%. This figure is essentially arbitrary, but projects are rarely 100% Python according to Github analysis. This is due to the presence of configuration files and documentation files in repositories. As the prototype RIFFLER tool only works on Python code, certain large projects are not yet suitable for analysis.

Projects that did not meet these requirements were discarded, and another project chosen uniformly at random, until sufficient projects are found.

Chapter 4

Optional Security Typing

Many prominent recent attacks have been memory leaks of sensitive data, such as Heartbleed, Meltdown and Spectre. Information Flow Control (IFC) type systems could have prevented them. Nonetheless, IFC type systems have not seen widespread industrial uptake. OAST, or Optional Security Typing, is a novel IFC that aims to facilitate uptake by allowing optional security annotations. The optional annotations are supported by information theoretic measures, resulting in a type system that incorporates testing into its decision procedure.

OAST combines gradual typing and quantified information flow in programs, gradualising IFC and allowing developers to incrementally add or remove security annotations. Local, incremental annotation avoids the upfront program-wide effort that may need to be revisited whenever the program changes.

Gradualising IFC is challenging. First, it can inhibit testing or refactoring, because code change may temporarily violate the security policy. Secondly, soundness forces the type checker to make conservative decisions about the security policy, thereby dynamically enforcing a policy that is possibly not intended by the developer. To meet both challenges, OAST leverages a new form of quantified

information flow called ranked information flow (RIF), to replace soundness with quantified risk.

4.1 Introduction

"Shall I be secure or insecure?"[1] Ideally, a secure program provably satisfies a security policy that circumscribes how information moves within the program. One type of policy often studied is a Denning-style lattice policy [66]. In this model, every variable has a security label, say, *public* and *private*. The labels form a lattice, *i.e.* $(\{public, private\}, \leq)$, with *flows* tracked by lattice operations over the labels. A variable marked *public* can flow into a *private* variable, but not vice-versa. Such a system is intuitive and inherently granular: one may label every single variable, or apply a label to an entire module. Each approach is equally expressive [163, 207]. In language-based security, many *information flow control* (IFC) languages utilise lattice systems to enforce *noninterference* (NI). Informally, noninterference means that confidential variables do not affect the publicly observable behaviour of a program (Section 4.3).

Writing a program that provably satisfies such a policy is very hard to do, so the answer to our question for most software is "insecure". This is insecurity through necessity or inability, not negligence. Noninterference is a very strict security policy: one way to relax noninterference is by increasing *expressiveness*, allowing more complex and realistic policies to be encoded [140]. Expressiveness includes the import dimension of *declassification*, allowing private data to be used in a public context in a controlled manner. Expressiveness and usability are not the same thing, however; this problem goes some way to explaining

[1] We allow ourselves the liberty of adapting a quote from Philip Wadler [211].

the relatively poor uptake of IFC in both research and industry. Encoding more realistic policies does not make it easier to perform the encoding, especially as developing functional properties must run in tandem with developing security properties. Doing both that the same time is more than most developers can manage. The complexities lead developers to handle security with *ad hoc* solutions that cannot be checked easily via static methods or human intervention. This is a major challenge for IFC languages: security counts in industry, so industry uptake counts.

An emerging approach to usability for type systems is gradual typing. This allows a developer to partially annotate a program [176], evolving the typing in conjunction with the program evolution. Applied to security typing, the hope is that security considerations are partially decoupled during functional properties. A partially typed gradual security system enforces NI at runtime. Some Gradual type systems enforce noninterference [93, 204], but fall foul of the dynamic part of the gradual guarantee [179]. Also, dynamic gradual typing comes with a runtime cost that is difficult to mitigate [194]. An alternative to gradual typing, optional typing, is increasingly found in industry. Languages with support for optional typing include PYTHON, TYPESCRIPT and HACK. Optional typing is a typing regime that forgoes the dynamically checked element of gradual typing. By doing do, a program stays performant, but sacrifices program-wide type safety. Such a sacrifice of program wide type safety is problematic for information security, as partial guarantees are no guarantees at all.

We present OAST, the first type system for IFC which mixes optional typing with information theoretic risk assessment and declassification. OAST allows a developer to decouple a program's security policy from its main functionality. To achieve this, it relaxes the static demands of NI in the form of an optional

system, *i.e.* one that has 'no effect on the runtime semantics of the programming language' [31]. To assist with annotations, both *what* and *where* to label, we use *ranked information flow*, RIF (Chapter 3) , a novel use of quantified information flow (QIF) [55] in IFC to order unannotated code by relative magnitude of flow. Every unannotated function in a program has its RIF calculated via random testing. RIF requires only a stable ordering of flows, rather than an estimate with a high degree of confidence. Once we have the ordering of annotation "holes", we have multiple options: we can annotate where flow is high, or we can further refine our RIF estimates. The ranking helps us prioritise leaks. RIF in a type system is also useful for declassification. While QIF is an expressive means of controlling declassification in a program [167], it has seen limited use in practice. This is largely due to the cost of its calculation. However, if we have a RIF ranking, we need only calculate a high quality estimate on the largest element in the ordering. If this comes in below our leakage tolerance threshold, then it is safe to assume that everything below it leaks a tolerable amount of information.

OAST offers, while typing is partial, information on what to annotate. The partiality of the type system allows it to model a simple declassification system, governed by a leakage threshold, calculated via RIF. As the number of annotations increases, the confidence in the NI guarantee likewise increases. This is because OAST satisfies the *confinement property*[2], a novel safety property formalised here for optional security type systems (Section 4.6). If there is a leak, it must come from a part of the code with no annotation. Confinement treats every fully typed subprogram as an independent program, which independently satisfies NI. The

[2]This confinement differs from the Bell-LaPadula Strong Star Property, which is sometimes informally also called confinement, in that it confines the origin of *leaks* to untyped regions, not writes only to the matching security level.

interface between modules is monitored via RIF. When typing is complete, there are no interfaces to monitor, and NI is satisfied. As the security properties of a program are separated from the functional properties, it is always possible to test a program without worrying that security errors trump functional errors. OAST considers a program as a collection of secure subprograms, communicating over potentially insecure channels; Bayesian information theoretic estimates track flows over this unknown channels and prioritise them by relative magnitude of flow. RIF provides useful feedback on where to start applying additional annotations. No gradual system that we know of provides the user with information on *what* and *where* to annotate.

"Shall I be secure or insecure?" We can be both; rather than *xor*, security can be reasoned about locally in a sound manner with fault prioritisation and localisation at the global level. With gradual typing one can be conservative and reject too much, or one can be liberal and except too much, thereby potentially leaking secure data. We propose a third way. Like the liberal approach, we allow leaks but use RIF to report and advise the developer at every stage, even post deployment.

Our principal contributions are:

- We present OAST, the first optional type system for information flow control;
- we show that RIF allows OAST to provide information on what to annotate and as a scalable form of information declassification;
- we introduce a new safety property for optionally typed languages, the confinement property, and prove that it holds for noninterference in OAST;

4.2 Background

OAST sits at the intersection of multiple different research strands. As this is the first work to sit in this intersection, few researchers have expertise in all the required disciplines. We present a short summary of basic concepts to facilitate reading this chapter. These concepts are covered extensively in Chapter 2.

Gradual type systems (Section 2.2) are a hybrid of static and dynamic type systems. They provide a spectrum of typing possibilities, from the complete absence of type annotations to complete typing information. Type safety is always guaranteed in a gradual type system: what cannot be resolved statically is resolved dynamically. The exact methods of this dynamic resolution are system dependent. In the research community, the gold standard for gradual systems is the Gradual Guarantee (Theorem 2.2.3), which states that adding or removing type annotations does not change program behaviour, apart from catching more or fewer errors at compilation time.

We distinguish gradual systems from optional systems. Optional systems do not satisfy the Gradual Guarantee. In fact, they have no additional runtime component. Systems such as MYPY for PYTHON and the type system of TYPESCRIPT are industry standard optional type systems. While they lack the guarantees of a gradual system, developers favour them for their lightweight analysis and because they do not penalise performance (Section 2.2.5). OAST is an optional type system for information security, the first of its kind.

Information Flow control (IFC) programming languages treat security as a first class concept, built into the language. This is commonly achieved via type systems (Section 2.1.2). A security policy they frequently seek to enforce is noninterference (Section 2.1.1). By Definition 2.1.1, program c satisfies noninterference

(more precisely, termination insensitive noninterference) if, for any memories μ and ν that agree on low variables, the memories produced by running c on μ and on ν also agree on low variables (provided that both runs terminate successfully).

Gradual IFC systems apply the logic of partial type information to this problem: classical noninterference assumes that all data has a known security level. Gradual IFC allows for data to have an unknown label, while still providing noninterference at runtime (Section 2.3). Resolving security labels at runtime in the face of incomplete information is hard (Section 2.3). As OAST does not have a runtime component to resolve labels, we propose a different technique: quantified information flow (QIF).

QIF measures information in a system (Section 2.3.2). It uses techniques from information theory and program analysis to quantify how much information passes through two observation points. At its heart, information theory uses *entropy* (Equation (2.1)) to quantify the information in a signal. Given the desire to know the effect of one variable (or signal) on another, we use the common information theoretic measure of mutual information (Equation (3.1)). This is a measure of dependence between two random variables. OAST does not use mutual information directly, but rather *FlowForward* (Definition 3.3.1), a normalised, asymmetric measure built on mutual information (Figure 3.3). As estimating *FlowForward* precisely is as expensive as estimating mutual information (Figure 3.2), we use the ranking method explored in Chapter 3 to reduce the testing cost.

4.3 Motivating Example

An optional security type system permits partial policy disclosure via labelling of variables and functions. This partial disclosure is checked statically. In a gradual type system, further checks are performed dynamically, during program

execution, to enforce NI. Toro *et al.* [204] demonstrate that this approach has various problems, in addition to the efficiency problems associated with gradual typing. A partially annotated program with no dynamic enforcement cannot be provably shown to satisfy NI. Our approach divides a program into statically known unknown parts, and interrogates their communication using techniques from information theory. This procedure is done during testing, not deployment. This procedure can be conducted to various degrees of accuracy and confidence, depending on requirements.

We consider a program in a simply typed λ-calculus, with a core of primitive types (Section 4.4). We assume two confidentiality levels, L = $\bot$ (public), H = $\top$ (secret) and the following functions, initially without any security labelling.

$$f \triangleq (\lambda x : Int. \ldots)$$
$$g \triangleq (\lambda x : Int. \ldots)$$
$$h \triangleq (\lambda x : Int.\ \lambda y : Int. \ldots)$$

With no policy disclosure for the type checker, any composition of functions that type checks on base types (those without a security annotation) also type checks in its security aspects. In the absence of security label information, the type checker might assume the $\star$ label, which trivially type checks. Worse, the type checker might assumes everything to be public, giving a correct, but spurious, guarantee of NI. A more sophisticated checker might make deterministic decisions on behalf of the developer. This is a problem for gradual type systems, as enforcing NI is not the same as enforcing the security policy desired by a developer. The developer has a requirement to provide sufficient information to the type checker

to disambiguate the intended policy from among the (possibly) many different NI-satisfying policies.

We add some security labels to a program written with these basic functions, and constant values. We assume constant values have label L. Let x and y be user provided values of unknown trustworthiness.

$$\lambda x : \text{Nat}.\, \lambda y : \text{Nat}\,.h_?(h_?(f_H(y), g_?(f_H(x))), 3)$$

We have colour coded the program to highlight how labels statically subdivide a program into parts about which we have sufficient knowledge, and insufficient. Code in red, if considered as *standalone* programs, is secure (Section 4.6.1). The H label of f serves to classify the user provided value of x. We do not, at this stage, know the security level of x itself. Note that it could still be the case that $f_H(x)$is illegal *w.r.t* to the intended policy: the provider of x may not have permission to call f.

If we consider the code fragment $g_?(f_H(x))$, we have an example of a secure value used in a context with unknown security value. The function g does not yet have a security label (here marked with ? for clarity). It is at this point that a gradual type system would use dynamic methods to enforce NI. We call this phenomenon, of secure code wrapped in an insecure context, a *chink*[3] (Section 4.6.1). Instead of dynamic enforcement at this point, OAST uses RIF (Chapter 3). It incorporates this as a testing phase.

First, we need to identify which functions require testing. In our example, we

[3]This term comes from the expression "a chink in the armour", which refers to a weakness in traditional metal armour. We are aware that in some countries, especially North America, this word is also used in a racist context. It is absolutely not our intention to cause offence in any way, but still regard this as the apposite term.

can see two values, $f_H(y)$ and $f_H(x)$, which have a confidential label. However, $f_H(x)$ appears in an insecure context, so OAST generates a test that calls $g_?$ with multiple integer values mocking the output of $f_H(x)$ (Section 4.5). We repeat this procedure for all chinks. In this simple program, we have three chinks, just enough to rank our unknowns by the magnitude of their flows. We want to do the smallest amount of testing possible in order to produce a trustworthy ranking. RIF uses *FlowForward* (Section 4.5), a real-value, in the interval $[0, 1]$, to perform its ranking. Underlying this value, however, is a Bayesian entropy estimation. We use this to test our confidence in our estimates.

Let us choose two chinks and say that chink 1 is $g_?(x)$ and chink 2 is $h_?(y, 3)$. RIF itself does not provide a guide to when to stop testing, so we look to the mutual information estimate provided by a Bayesian estimator. The NSB (Nemenman-Schafee-Bialek) algorithm [144] returns the mean and standard deviation of its distribution for its estimate. Given the available parameters, we assume a normal distribution. We use both the standard deviation and the *Wasserstein distance* [219] as stopping parameters, rather than the experimental evidence of stabilisation as provided in Section 3.5. We detail this in Section 4.5.

Figure 4.1 plots the two estimates of MI from the two chinks in our program. After just 100 tests, the estimates are unlikely to be sufficient to reach conclusions about the ordering. The two curves share a great deal of overlap, too much to be comfortable in stating that they are definitely drawn from different distributions and that there is a statistically significant different between the two estimates. Depending on how certain we wish to be, we can set our parameters to request extra tests. Ones sees the effect of this in Figure 4.2, where we can be more certain that we are seeing is different information behaviour, and not just error due to our sampling regime. We explore this in more detail in Section 4.5.

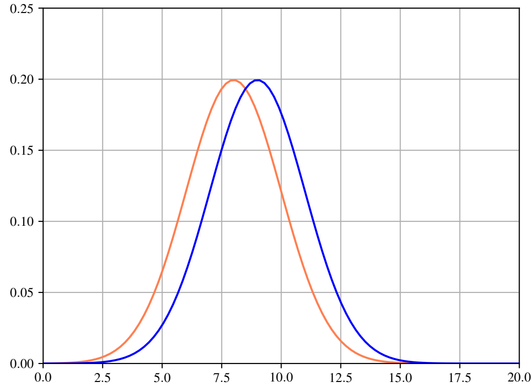

Figure 4.1: Mutual information estimates after 100 tests each. The standard deviations of both distributions are the same, but the Wasserstein distance is only 0.0002. These are close, but the standard deviation threshold has not been reached, so testing continues to see if they really have the same ranking, or represent distinct points in the ordering of chinks.

4.4 OAST Language

To reify optional security typing with ranked information flow, we present λ_{OAST}, a pure functional language with eager semantics. λ_{OAST} is an instantiation of the OAST concept. λ_{OAST} is a simply typed λ-calculus extended with a optional security labels.

4.4.1 Core Language

Figure 4.3 presents the abstract syntax of λ_{OAST}. The typing rules were derived using the *AGT* method of Garcia and Tanter [91]. *AGT* uses principles drawn from abstract interpretation, creating a formal method for deriving gradual type systems from existing languages. We only require one half of the *AGT* method, as we do not require gradualised dynamic semantics. As we have used *AGT*, the

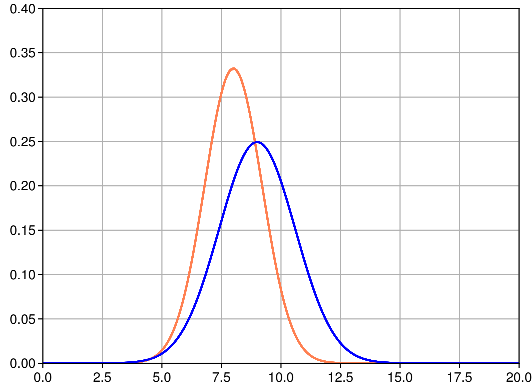

Figure 4.2: After 100 more tests of each function, we now have narrower standard deviations (1.2 and 1.6) with similar means. Our confidence has increased in our ranking. The Wasserstein distance is now 0.01, meaning it is less likely that we are looking at sampling from the same distribution. We take these to represent separate points in the ordering.

static rules of λ_{OAST} share features with the same authors presentation of a simply typed, security λ-calculus, $\lambda_{\widetilde{SEC}}$ [93]. $\lambda_{\widetilde{SEC}}$ itself is a gradualised counterpart to Zdancewic's λ_{SEC} [223]. We provide the syntax of λ_{OAST} in Figure 4.3. This gives the calculus after transformation to include security labels. We do not detail the procedure for gradualising a non-gradual calculus.

The most salient difference between λ_{OAST} and a simply typed λ-calculus is that every type also bears a security label, including annotations on the 'arrow' type. The simplest function type consists of three security labels $x_{\ell} \rightarrow_{\ell'} y_{\ell''}$. The overall security label of this type is the iterated join over its labels, *i.e.* $((\ell \sqcup_{\star} \ell') \sqcup_{\star} \ell'')$ where $\sqcup_{\star}$ is the gradual join defined for this lattice (Section 4.4.2).

$$\ell \in \textsc{Concrete Label},\ S \in \textsc{LabelledType},\ x \in \textsc{Var},\ b \in \textsc{Bool},\ n \in \textsc{Nat}$$
$$t \in \textsc{Term},\ \otimes \in \textsc{BoolOp},\ \oplus \in \textsc{NatOp}$$
$$\neg \in \textsc{UnaryOp},\ \Gamma \in \textsc{Var} \rightharpoonup \textsc{LabelledType},\ r \in \textsc{Unlabelled Value}$$
$$v \in \textsc{Value}$$

$$
\begin{array}{lcl}
\ell & \rightarrow & \bot \mid \top \\
\ell_\star & \rightarrow & \ell \cup \star \\
S & \rightarrow & \mathrm{Bool}_{\ell\star} \ \mid\ \mathrm{Nat}_{\ell\star} \ \mid\ S \rightarrow_{\ell\star} S \\
b & \rightarrow & \mathrm{true} \ \mid\ \mathrm{false} \\
n & \rightarrow & \mathrm{zero} \ \mid\ \mathrm{succ}\, n \\
r & \rightarrow & b \ \mid\ n \ \mid\ \lambda x : S.\, t \\
v & \rightarrow & r_{\ell_\star} \\
t & \rightarrow & v \mid t\ t \mid t \otimes t \mid \neg t \mid t \oplus t \mid \mathrm{if}\ t\ \mathrm{then}\ t\ \mathrm{else}\ t
\end{array}
$$

Figure 4.3: λ_{OAST} is an extension of a simply typed λ-calculus to include optional security labels and a fixpoint term to allow for general recursive computations. The security label variable is denoted ℓ, where ℓ is either $\top$ or $\bot$. Only security labels have optional values; we assume that base type values are known.

4.4.2 Security Labels

λ_{OAST} features a $\{\bot, \top\}$ security label model extended with a $\star$ (any) label. While $(\{\bot, \top\}, \sqsubseteq)$ provides an easily defined two point lattice, the addition of a $\star$ necessitates changes to the calculation of lattice operations. In general, the addition of a $\star$ to a lattice of security labels breaks the lattice requirements. Thus, it is an abuse of terminology to refer to gradual security lattices. Indeed, $\star$ breaks the requirements to even be a partial order, since security label consistency (Figure 4.6) is not transitive. We detail gradual joins and meets in Figure 4.4 and Figure 4.5. One sees that $\star$ 'sits in the middle': for gradual joins, it dominates $\bot$, erasing that

$$\top \sqcup_\star \star = \star \sqcup_\star \top = \top$$

$$\ell_\star \sqcup_\star \star = \star \sqcup_\star \ell_\star = \star \text{ if } \ell_\star \neq \top$$

$$\ell_1 \sqcup_\star \ell_2 = \ell_1 \sqcup \ell_2 \textit{ otherwise}$$

Figure 4.4: Security label join. The join over concrete labels, denoted simply $\sqcup$, is standard.

$$\bot \sqcap_\star \star = \star \sqcap_\star \bot = \bot$$

$$\ell_\star \sqcap_\star \star = \star \sqcap_\star \ell_\star = \star \text{ if } \ell_\star \neq \bot$$

$$\ell_1 \sqcap_\star \ell_2 = \ell_1 \sqcap \ell_2 \textit{ otherwise}$$

Figure 4.5: Security label meet. The meet over concrete labels, denoted simply $\sqcap$, is standard.

label and replacing it with itself, but is erased in turn by $\top$. The inverse holds for gradual meets.

During type checking, we often need to update the value of the security label of a term. The return type of an binary operation over integers, for example, is the join of its operands. As an example, if $x : \text{Nat}_{\ell_1}$ and $y : \text{Nat}_{\ell_2}$ then the term $x+y$ has the type $x+y : \text{Nat}_{\ell_1 \sqcup \ell_2}$ (as $+ : \bot \to \bot \to \bot$). To perform these updates, we define a label stamping function over the shape of the type (Figure 4.7).

In λ_{OAST}, base types are types with security labels erased; that is, there is a function *erase* that takes $\text{Nat}_\bot \to_\bot (\text{Bool}_\bot \to_\bot \text{Nat})$ to $\text{Nat} \to (\text{Bool} \to \text{Nat})$. There is no implicit $\star$ label in the second signature: it is a type in the underlying λ-

$$\frac{}{\ell \approx_\ell \ell} \quad \frac{}{\ell \approx_\ell \star} \quad \frac{}{\star \approx_\ell \ell}$$

Figure 4.6: Label consistency: $\star$ is consistent with all security labels.

$$S_{\ell_\star} \sqcup_\star \ell'_\star = S_{(\ell_\star \sqcup_\star \ell'_\star)}$$
$$(S_1 \rightarrow_{\ell_\star} S_2) \sqcup_\star \ell'_\star = S_1 \rightarrow_{(\ell_\star \sqcup_\star \ell'_\star)} S_2$$

Figure 4.7: Label stamping updates the security label of a term during type checking. It does not affect the base type, as these are not gradualised in OAST.

calculus, not in λ_{OAST}. A concrete type in λ_{OAST} is one about which everything is known, both its base type and the concrete value of all security labels. Base types are not gradual and are checkable in a separate pass. $\text{Nat}_\perp \rightarrow_\perp \text{Nat}_\top$ is a concrete type, while $\text{Nat}_\star \rightarrow_\perp \text{Nat}_\top$ is not.

The labelling system of λ_{OAST} is *fine-grained* in the sense that every function and function parameter takes a security annotation. This is in contrast to more coarse-grained labelling approaches in which entire 'blocks' of code have one security label (Section 2.2.7). Hard coded literals, such as booleans and integers, default to public. This conforms with a standard assumption of NI attacker models, that the source code is available [165]. Every type in λ_{OAST} has a slot for security label, including function types (Figure 4.3).

Definition 4.4.1 (Concrete Type). A concrete type is a base type labelled with all

concrete security labels ℓ.

$$C \in \textsc{Concrete Security Type}$$

$$C ::= \text{Bool}_\ell \mid \text{Nat}_\ell \mid \; C \rightarrow_\ell C \mid$$

$$\frac{t_1 : \text{Bool}_{\ell^1_\star} \qquad t_2 : \text{Bool}_{\ell^2_\star} \qquad \ell^1_\star \approx_\ell \ell^2_\star}{\text{Bool}_{\ell^1_\star} \approx_t \text{Bool}_{\ell^2_\star}} \qquad \frac{t_1 : \text{Nat}_{\ell^1_\star} \qquad t_2 : \text{Nat}_{\ell^2_\star} \qquad \ell^1_\star \approx_\ell \ell^2_\star}{\text{Nat}_{\ell^1_\star} \approx_t \text{Nat}_{\ell^2_\star}}$$

$$\frac{t_1 \rightarrow_{\ell^1_\star} t_2 \qquad t_3 \rightarrow_{\ell^2_\star} t_4 \qquad t_1 \approx_t t_3 \qquad t_2 \approx_t t_4 \qquad \ell^1_\star \approx_\ell \ell^2_\star}{t_1 \rightarrow_{\ell^1_\star} t_2 \approx_t t_3 \rightarrow_{\ell^2_\star} t_4}$$

Figure 4.8: Type consistency rules for λ_{OAST}.

We replace type equality with type consistency (Figure 4.8). We define a consistency relation for the static rules in the manner of sound gradual typing. The key observation is that type consistency, unlike equality, is not transitive. The 'fuzzy' nature of $\star$ does not fit into an equivalence relation: $\text{Nat}_\bot \approx_t \text{Nat}_\star$ and $\text{Nat}_\star \approx_t \text{Nat}_\top$ but, clearly $\text{Nat}_\bot \approx_t (\text{Nat}_\star \approx_t \text{Nat}_\top)$ does not imply that $\text{Nat}_\bot \approx_t \text{Nat}_\top$. Base types use the standard definition of type equality, consistency is relevant to labels.

Figure 4.9 gives the typing rules for λ_{OAST}. To reduce notational clutter a little, we let S range over security types (base types with security label either $\bot, \top$ or $\star$), without making the presence of $\star$ explicit. The most interesting rule is function application, SAPP. The security level of function application is the join of the lambda abstraction's label, and the right hand side of the function type. In effect, as long as the argument is a subtype of the formal parameter, its label does not affect the output label. λ_{OAST} has a flow-insensitive static system in the sense

$$\boxed{\Gamma \vdash t : S}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : S} \ (\text{SX}) \qquad \frac{}{\Gamma \vdash b_{\ell_\star} : \text{Bool}_{\ell_\star}} \ (\text{SB}) \qquad \frac{}{\Gamma \vdash n_{\ell_\star} : \text{Nat}_{\ell_\star}} \ (\text{SN})$$

$$\frac{\Gamma, x : S_1 \vdash t : S_2 \qquad \ell_\star}{\Gamma \vdash (\lambda x : S_1 . t)_{\ell_\star} : S_1 \rightarrow_{\ell_\star} S_2} \ (\text{S}\lambda) \qquad \frac{\Gamma \vdash t_1 : \text{Bool}_{\ell_\star^1} \qquad \Gamma \vdash t_2 : \text{Bool}_{\ell_\star^2}}{\Gamma \vdash t_1 \otimes t_2 : \text{Bool}_{(\ell_\star^1 \sqcup_\star \ell_\star^2)}} \ (\text{S} \otimes)$$

$$\frac{\Gamma \vdash t : \text{Bool}_{\ell_\star}}{\Gamma \vdash \neg t : \text{Bool}_{\ell_\star}} \ (\text{S} \neg) \qquad \frac{\Gamma \vdash t_1 : \text{Nat}_{\ell_\star^1} \qquad \Gamma \vdash t_2 : \text{Nat}_{\ell_\star^2}}{\Gamma \vdash t_1 \oplus t_2 : \text{Nat}_{(\ell_\star^1 \sqcup_\star \ell_\star^2)}} \ (\text{S} \oplus)$$

$$\frac{\Gamma \vdash t_1 : S_{11} \rightarrow_{\ell_\star} S_{12} \qquad \Gamma \vdash t_2 : S_2 \qquad S_2 <: S_{11}}{\Gamma \vdash t_1 \ t_2 : (S_{12 \sqcup_\star \ell_\star})} \ (\text{SAPP})$$

$$\frac{\Gamma \vdash t : \text{Bool}_{\ell_\star} \qquad \Gamma \vdash t_1 : S_1 \qquad \Gamma \vdash t_2 : S_2 \qquad S_1 \tilde{\triangledown} S_2}{\Gamma \vdash \text{ if } t \text{ then } t_1 \text{ else } t_2 : (S_1 \tilde{\triangledown} S_2)_{\sqcup_\star \ell_\star}} \ (\text{SIF})$$

Figure 4.9: OAST inference rules; Figure 4.10 defines $\tilde{\triangledown}$ and Figure 4.11 defines $<:$. S ranges over security types, *i.e.* base types labelled with either a concrete label or $\star$.

that "flow-insensitivity uses a single abstraction (in this case a single security level) to represent each variable in the program" [108]. We do not yet consider a system with type-level declassification; a $\top$ variable remains high throughout type checking: it cannot be directly re-labelled $\bot$. It can, however, have its label effectively erased via a $\star$-typed function (Section 4.6.1). Such a function introduces a chink into the program, which becomes subject to RIF analysis (Section 4.5).

$$\boxed{S \tilde{\triangledown} S, S \tilde{\wedge} S}$$

$$\begin{array}{ll}
\tilde{\triangledown} : \textsc{Type} \times \textsc{Type} \rightharpoonup \textsc{Type} & \tilde{\wedge} : \textsc{Type} \times \textsc{Type} \rightharpoonup \textsc{Type} \\
\text{Bool}_{\ell_\star} \tilde{\triangledown} \text{Bool}_{\ell'_\star} = \text{Bool}_{\ell_\star \sqcup_\star \ell'_\star} & \text{Bool}_{\ell_\star} \tilde{\wedge} \text{Bool}_{\ell'_\star} = \text{Bool}_{\ell_\star \sqcap_\star \ell'_\star} \\
\text{Nat}_{\ell_\star} \tilde{\triangledown} \text{Nat}_{\ell'_\star} = \text{Nat}_{\ell_\star \sqcup_\star \ell'_\star} & \text{Nat}_{\ell_\star} \tilde{\wedge} \text{Nat}_{\ell'_\star} = \text{Nat}_{\ell_\star \sqcap_\star \ell'_\star} \\
(S_{11} \rightarrow_{\ell_\star} S_{12}) \tilde{\triangledown} (S_{21} \rightarrow_{\ell'_\star} S_{22}) = & (S_{11} \rightarrow_{\ell_\star} S_{12}) \tilde{\wedge} (S_{21} \rightarrow_{\ell'_\star} S_{22}) = \\
\quad (S_{11} \tilde{\wedge} S_{21} \rightarrow_{\ell_\star \sqcup_\star \ell'_\star} (S_{12} \tilde{\triangledown} S_{22}) & \quad (S_{11} \tilde{\triangledown} S_{21}) \rightarrow_{\ell_\star \sqcap_\star \ell'_\star} (S_{12} \tilde{\wedge} S_{22}) \\
S \tilde{\triangledown} S, \text{ undefined otherwise} & S \tilde{\wedge} S, \text{ undefined otherwise}
\end{array}$$

Figure 4.10: Joins and meets on types; Figure 4.9 uses the operators defined here. These definitions are mutually recursive.

$$\boxed{S <: S}$$

$$\frac{\ell_\star \leq \ell'_\star}{\text{Bool}_{\ell_\star} \leq: \text{Bool}_{\ell'_\star}} \qquad \frac{\ell_\star \leq \ell'_\star}{\text{Nat}_{\ell_\star} \leq: \text{Nat}_{\ell'_\star}} \qquad \frac{S'_1 \leq: S_1 \qquad S_2 \leq: S'_2 \qquad \ell_\star \leq \ell'_\star}{S_1 \rightarrow_{\ell_\star} S_2 \leq: S'_1 \rightarrow_{\ell'_\star} S'_2}$$

Figure 4.11: λ_{OAST} subtyping rules.

The subtyping rules in Figure 4.11 extend those in Zdancewic [223] and Garcia and Tanter [91]. It is perhaps a bit of a misnomer to refer to these as subtype rules in that $\tau_\bot$ is not a *subtype* of $\tau_\top$; rather, it is a type indexed by a security label. It seems reasonable, however, to stay with the accepted terminology if, as in this case, it is does not lead to any confusion. The $\star$ security label comes anywhere in the subtype relation, *i.e.* $\star \leq \bot$ and $\top \leq \star$.

Definition 4.4.2 (Concrete Security Labels and Security Labels)**.** A *concrete security*

label ℓ is a point ($\{\bot, \top\}$) in the label lattice $\mathscr{L}$.
A *security label* $\ell_\star$ is either a concrete label $\ell \in \mathscr{L}$ or the unknown label $\star$.

That is, a security label can be any element drawn from the set $\mathscr{L} \cup \star$ of labels, whereas a concrete label explicitly excludes $\star$; it is a statically declared value drawn from the set $\mathscr{L}$, where $\star \notin \mathscr{L}$. In the case of λ_{OAST}, this means a concrete label is either $\top$ or $\bot$, which we map to private and public terms. Provided the partial ordering $\bot \leq \top$ we construct, without loss of generality, a simple security lattice. In this (concrete) lattice, the only disallowed flow is from $\top$ to $\bot$, *i.e.* $\top \nleq \bot$.

Operational semantics for λ_{OAST} are standard as for a simply typed λ-calculus. We present them in the form of big-step semantics, Figure 4.12, which is sufficient for proving *Termination Insensitive Noninterference*.

4.5 Ranking Flows for Security

Measuring the information flowing through a system is powerful: QIF offers an alternative, systematic, method for relaxing noninterference. QIF allows reasoning about information flow at a very fine granularity, but is very expensive to compute exactly (Section 4.7). To address this problem, we utilise *Ranked Information Flow*, a novel approach to reducing the cost of QIF via Monte Carlo methods. Rather than waiting until there is a single high quality estimate, RIF relies on the fact that the ordering of flows by magnitude requires fewer samples. The intuition why is simple: much of the time spent in generating a high quality estimate is in the refinement process, when the provisional estimate is already "in the ballpark". Ballpark estimates are good enough for an ordering, assuming that the risk of the two estimates flipping place is sufficiently low. RIF produces a reliable information theoretic ordering that experimental evidence shows can be computed more efficiently. We extend the RIF approach to strengthen confidence

$\boxed{t \Downarrow v}$

$$\frac{}{v \Downarrow v} \quad (\textsc{Eval-Val}) \qquad \frac{e_1 \Downarrow b_1 : Bool_{\ell_1} \qquad e_2 \Downarrow b_2 : Bool_{\ell_2}}{e_1 \otimes e_2 \Downarrow (b_1 [\![\otimes]\!] b_2)_{(\ell_1 \sqcup \ell_2)}} \quad (\textsc{Eval-BinOpB})$$

$$\frac{e_1 \Downarrow n_1 : Nat_{\ell_1} \qquad e_2 \Downarrow n_2 : Nat_{\ell_2}}{e_1 \oplus e_2 \Downarrow (n_1 [\![\oplus]\!] n_2)_{(\ell_1 \sqcup \ell_2)}} \quad (\textsc{Eval-BinOpA})$$

$$\frac{e_1 \Downarrow b_1 : Bool_{\ell}}{\neg e_1 \Downarrow [\![\neg]\!] b_{1_\ell}} \quad (\textsc{Eval-UnOpBool})$$

$$\frac{e \Downarrow b_{\ell} \qquad e_1 \Downarrow v_{\ell'}}{\text{if } e \text{ then } e_1 \text{ else } e_2 \ \Downarrow v_{\ell \sqcup \ell'}} \quad (\textsc{Eval-Cond}_1)$$

$$\frac{e \Downarrow b_{\ell} \qquad e_2 \Downarrow v_{\ell'}}{\text{if } e \text{ then } e_1 \text{ else } e_2 \ \Downarrow v_{\ell \sqcup \ell'}} \quad (\textsc{Eval-Cond}_2)$$

$$\frac{e_1 \Downarrow (\lambda x : s.e)_{\ell} \qquad e_2 \Downarrow v \qquad e\{v/x\} \Downarrow v'_{\ell'}}{e_1 \ e_2 \Downarrow v'_{(\ell \sqcup \ell')}} \quad (\textsc{Eval-App})$$

Figure 4.12: Big step operational semantics for λ_{OAST}.

in the ordering.

Ranking directly over mutual information is problematic (Section 3.3). Mutual information, $I(X;Y)$, is the information measure given by

$$I(X;Y) = \sum_{x \in X, y \in Y} p(x,y) \log_2 \left(\frac{p(x,y)}{p(x)p(y)} \right)$$

As a flow of 1 bit from a boolean valued function is more than a flow of 3 bits from an integer valued, ranking chinks via their mutual information would imply that the 3 bit flow was always worse than the 1 bit flow, and this might not be the case. We do the final ranking and estimation using *FlowForward* (Definition 3.3.1), but change the testing termination condition of entropy.

$$FlowForward(X,Y,\delta) = \delta\left(1 - \frac{H(X) - I(X;Y)}{H(X)}\right)$$

FlowForward and mutual information can be estimated through trial runs of the software under test (SUT). As Chothia *et al.* [49] show in their work on the *LeakWatch* tool for Java, a large number of samples is often required to ensure that the estimated mutual information $\hat{I}(X;Y)$ is close to the real mutual information $I(X;Y)$. This large number is 'many more samples than the product of the number of secret to observable values' [49]. As a result, when the secret's I/O domains are large, the number of trial runs becomes problematic. To reduce some of the problems associated with a small sample size, the PYTHON RIF calculator, RIFFLER (Section 3.4) uses the Bayesian NSB entropy estimator [144, 143]:

$$P(\alpha) = \frac{1}{\log K}(K\psi_1(K\alpha+1) - \psi_1(\alpha+1)) \tag{4.1}$$

where ψ_1 is the *trigamma* function [214] and K is the number of samples. The virtue of using a Bayesian approach is that it returns a distribution, not just a point estimate. We utilise this fact to introduce a means to define a termination condition for ranking, something absent from RIFFLER (Section 3.4). Confidence in ranking is controlled via two parameters: the standard deviation of the estimate, and the Wasserstein distance between two estimates, *i.e.* probability distributions.

This latter is more commonly known as *Earth mover's distance* in computer science:

$$\mathrm{EMD}(P,Q) = \inf_{\gamma \in \prod(P,Q)} \mathbb{E}_{(x,y)\sim\gamma}[\|x - y\|] \tag{4.2}$$

where P and Q are the two distributions being compared, and $\prod(P,Q)$ is the set of all joint distributions whose margins are P and Q. In short, if the deviation around the estimate is sufficiently small, then we use the Wasserstein distance to decide whether the two estimates have the same rank or differ.

Obtaining chinks To use this approach for a program, we first need to generate a list of chinks from that program. One obtains these via the static typing information. In the event that there is only one chink, no ordering is possible. In this case, it is reasonable simply to test until a high quality estimate is achieved. In the event that there is insufficient information to detect the definite presence of chinks (*i.e.* there are only $\perp$ and $\star$ labels), the user can, of course, opt to test all functions anyway. The requirement to generate a list of chinks for testing is largely to reduce resource usage.

Once we have obtained this list, we need to generate tests for the outer function, the entry point to our chinks. Such entry points will always have the $\star$ security annotation, but contain within them a $\top$ computation. We test the chinks by generating random inputs of the appropriate types, saving both inputs and outputs in a database. Each function is tested a minimum of n times, with n being a user provided parameter. After every n tests, we calculate the ranking and analyse it in accordance with Algorithm 1. This algorithm is an extension to the basic testing approach used by RIFFLER.

Algorithm 1: Modified RIF algorithm that includes the twin stopping criteria of standard deviation (std_target) and wasserstein distance (wsd_target). New batches of n input/output pairs are added until the stopping predicates are satisfied.

```
   Input: C = [chink], std_target, wsd_target, n
   Output: [(FlowForward, chink)]
 1 L ← []
 2 foreach c ∈ C do
 3     st = std + 1                            // set while condition to true
 4     while st > std do
 5         ss ← test(c, n)                      // call c n times and collect output
 6         m, st ← NSB(ss)                      // calculate mean and standard deviation
 7     end
 8     L.append((m, st, c))                     // If st is sufficiently small, save
        details
 9 end
10 R ← []
11 L ← sort(L)                                  // sort L by m estimate
12 for i ∈ [0, len(L) -1] do
       /* pairwise comparison of Wasserstein Distance in ordered
          list */
13     d ← wasserstein(L[i], L[i + 1])
14     if d ≥ wsd_start then
15         f ← FlowForward(L[i].m)
16         R.append((f, c))
17     end
18 end
19 return R
```

```python
from cryptography.fernet import Fernet
def get_message()_L -> bytes_H:
    message = input('Enter Secret Message: ')
    return message.encode('utf-8')

key : bytes_H = Fernet.generate_key()
f : Fernet_L = Fernet(key)

token = f.encrypt(get_message())

out = f.decrypt(token)
```

Listing 4.1: An embedding of OAST into Python. Security annotations are appended to normal Python type hints. These are type checked with a separate analysis pass.

4.5.1 A Python Prototype

OAST's combination of theorem-based validation (Definition 4.6.1 and information theoretic reasoning over information flow is unusual. The proof of confinement is relatively easy for our core language (Figure 4.3), due to its purity. We examine in this section a shallow embedding of the OAST approach in Python, highlight some of the advantages and attendant difficulties, and also demonstrating the savings of RIF over more exhaustive statistical methods.

The Python *cryptography* module provides cryptographic primitives and recipes for Python developers. It does not feature any type hints. No existing Python static type checker known to us allows the user to create new first class types, as required for the custom logic of optional security types. One can rely on

dynamic methods, *i.e.* refinement types and assertions, but these have precisely the runtime implications that we wish to avoid with OAST. Barring an extension to *mypy* or other type checker, we embed an optional system in OAST via type annotations suffixed with `_L` and `_H`. We use a simple static analysis to generate a callgraph, and then type check the security components only. These security suffixes can erasable, so that the program can also be type checked with a standard Python tool.

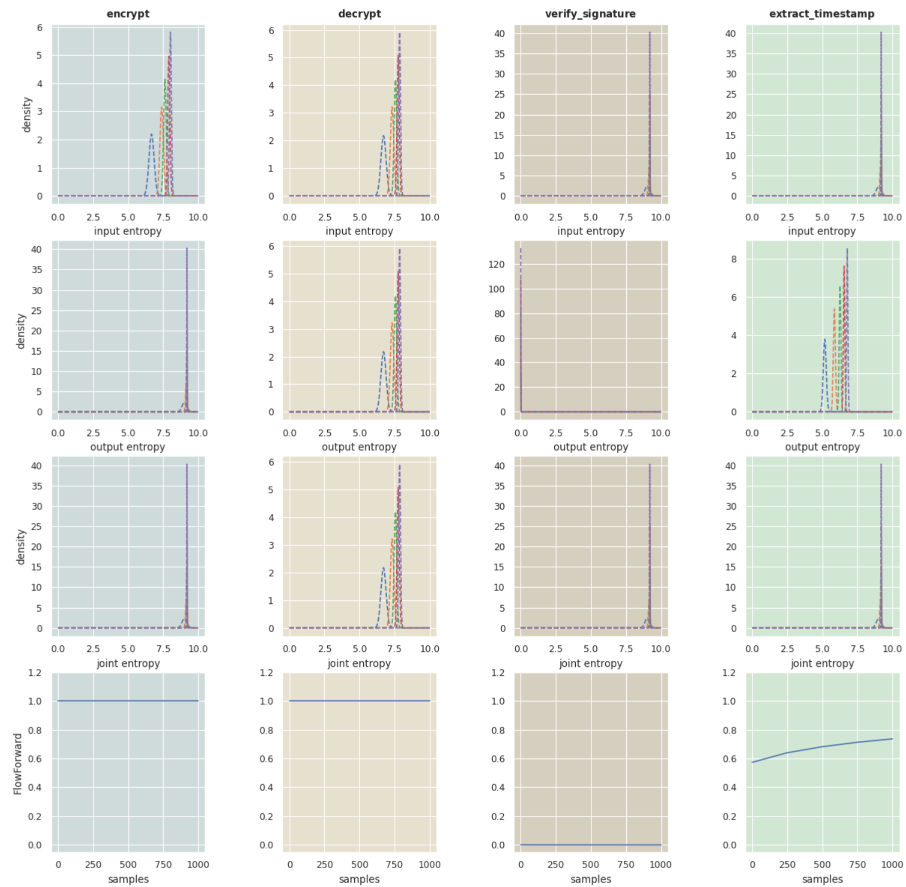

Figure 4.13: Four chinks in a partially annotated Python module. The quality of the estimates improves with time. By 600 samples, there is sufficient confidence in the ranking to halt. The function `verify_signature` has a low *FlowForward*. It is unlikely to create a problematic leak. `extract_timestamp` passes nearly all its information: RIF highlights this function for further investigation. Both `encrypt` and `decrypt` have high *FlowForward*, as expected from their functionality. Due to the normalising properties of *FlowForward*, the ranking is stable in this case after just 100 samples.

We show some example code in Listing 4.1. This code has "obvious" annotations applied: the type signature of `get_message()`, for example, means that it is a public function, usable at any security level (as shown by the $_L$ on the parentheses). The function returns a bytestring that must be secret, as it is the message to be encoded. We applied annotations to part of the cryptography module `Fernet`. This produced 4 chinks, which were then tested, with the results in Figure 4.13. We relied on manually written tests using the HYPOTHESIS property-based testing library [126] to generate random input values. It is important to note that these chinks are not necessarily leaks in the cryptography module *per se*: policy in the module is implicit and both `encrypt` and `decrypt` are expected to let information pass through them. The key is, of course, that the information should be distorted and not easily recoverable by an attacker.

PYTHON does not enforce any privacy policy on methods or functions, so all and any functions are available to a developer. It is, of course, good practice that library code should behave in secure and predicable ways: knowing where the chinks are, and how much they *FlowForward* information allows the library author to apply appropriate annotations, making it more difficult for a library user to misuse functionality.

We set k in the Algorithm 1 at 100, and `target_std` at 0.1. These numbers can be changed depending on resource budget. The Wasserstein threshold was likewise set at 0.1. If two estimates have a Wasserstein distance less that 0.1, then they are considered to have the same *FlowForward*, and given the same ranking. Again, this can be set at user discretion. The values provided here were derived from small scale studies: further work is required to provide good default values for both deviation and distance.

The most interesting chinks in Figure 4.13 are obviously `encrypt` and `decrypt`.

They both have high information flow. At 500 samples, the standard deviations for both `decrypt` and `encrypt` are sufficiently low to meet the stopping condition for all three estimated entropies (input, output and joint). Likewise, the wasserstein distances between the different estimates provides reason to trust the *FlowForward* estimate: for example, the distance between the input estimates of `_verify_signature` and `extract_timestamp` at 400 samples is 0.15, and the *FlowForward* estimates, at no point, approach each other over the testing run. In order to further reduce the work required, we average the distance over the 3 different entropy estimates. The value of *FlowForward* is clear from Figure 4.13: the ranking is clear from just 100 samples and does not alter, even though the individual entropy estimates are mobile. When there are more chinks to explore, the benefit of increased testing is more apparent. The *FlowForward* for `extract_timestamp`, for example, can be seen to be rising slowly, before starting to flatten. If there were other chinks in the same range, then the order might change slightly.

The importance of this approach is now clear. In any current gradual security type system, it would not be possible to type `decrypt`. It receives obviously confidential data, but no existing system yet allows for declassification: `decrypt` would fail at runtime if left unannotated, even though it is not a security risk. OAST not only draws our attention to the function, it gives the user confidence that it may be ignored for typing. The absence of runtime enforcement is not a negative if the behaviour of the function is already understood from an information theoretic perspective. Likewise, even though `_verify_signature` handles confidential information, its output is public, but uninformative. This would not be possible to type in any gradual system without declassification. Given the complex semantics that already exist for gradual security typing *without* including declassification, this highlights the elegance of the OAST approach.

4.6 Noninterference and Confinement

"How small a chink lets in how dire a foe." — Ants, William Empson [75]

OAST does not try to satisfy the dynamic component of the gradual guarantee. By construction, as proved by Garcia *et al.* [91], a sound type system lifted via the *AGT* method satisfies the static part of the gradual guarantee (Section 2.2.2). This static part states that if $\Gamma_1 \vdash c_1, \Gamma_1 \sqsubseteq \Gamma_2$ and $c_1 \sqsubseteq c_2$, then $\Gamma_2 \vdash c_2$. This condition requires that adding (or removing) *correct* type annotations does not affect the correctness of the type analysis. Essentially, by removing annotations, we are asking the language to trust the developer, but any that we add can be rigorously checked for correctness. The dynamic part of the gradual guarantee is much more difficult to satisfy; this requires that any typing derivation constructed during runtime to correct and enforceable. This has been shown to be very hard for information flow languages [8].

Given that OAST does not include a dynamically type checked element, we only prove satisfaction of NI for fully annotated programs (Section 4.9.1). It is not the case however, that we cannot say anything useful about the static guarantees of a partially annotated program. We introduce a useful safety property which we call the *confinement property* in Definition 4.6.1. Informally, this says that, if our program leaks, then the leak cannot be in a fully typed fragment; the problem must occur somewhere where there are no annotations. The confinement property formalises the notion that unannotated code need not be sound, but fully annotated code is trustworthy. This means that code within annotated regions can be effectively ignored during IFC debugging. This is a practical and useful property for any optional type system to possess and adds strength to type systems that satisfy only the static requirements of the gradual guarantee.

The intuition behind *confinement* is simple: we decompose the program into fully typed fragments and check that those satisfy NI. These fully typed fragments are essentially mini-programs, a property we possess due to the purely functional nature of λ_{OAST}. We type check these mini-programs and either error, or record the conclusion of the typing derivation. Mini-programs which are typed as $\bot$ do not contribute to program risk. Mini-programs typed $\top$, if used within an untyped context, are chinks and assessed for risk.

We formalise this notion as follows: let *sts* be a non-optional information flow type system for some simple λ calculus, such that $\vdash t : \tau_\ell$ is a typing judgement in *sts*. Semantics in the language of *sts* must satisfy the standard definition of type preservation. Let $sts_\star$ be *sts* extended with $\star$, such that $\vdash_\star t : \tau_{\ell_\star}$ is a judgement in $sts_\star$. $\Gamma_\star$ denotes the environment of $sts_\star$ whereas Γ denotes the environment for *sts*.

Definition 4.6.1. Confinement Property
$\forall t \in \textsc{Term},\ \text{if}\ \Gamma_\star \vdash_\star t : \tau \Downarrow v : \tau \wedge \Gamma \not\vdash t : \tau\ \text{then}\ \exists x, x \in FV(t),\ \Gamma_\star(x) = \star \wedge \Gamma(x) = \top$

where $FV(t)$ are the free variables of t and $\top$ is the top element in the security lattice.

This states that, if a term t type checks under optional labels, but does not type check when concretely labelled, then the error must be found in a subterm of t, say t', that has the $\star$ label and itself contains a subterm of security label $\top$. Confinement is a desirable property of an optional security type system. It is especially useful when wishing to reason safely about a subset of a statically unsafe type system. This characterisation is rather different from sound gradual typing, which is a dynamically safe extension to a safe underlying type system. Blame, in the Wadler sense [212], is not possible in the optional typing setting, as optional type systems do not, by definition, have a runtime component and

therefore cannot track blame.

For our language (Section 4.4.1), security annotations confine *Termination Insensitive Noninterference* [165]. Confinement means that any information leak in the program originates in an unannotated area. The word "originates" perhaps needs some clarification: while the information leaked may come from a typed fragment (it is the presence of a $\top$ label that denotes it as a leak rather than just an information flow), the leak itself is the point where the information becomes publicly visible. If an annotated region of code is typeable (*i.e.* is accepted by the type checker), then we can safely ignore it when debugging and refactoring, even if it is the source of the information under threat.

Given that the presentation of λ_{OAST} is a purely function language, the proof that it confines *Termination Insensitive Noninterference* and satisfies the confinement property is relatively straightforward. More complex, possibly stateful, languages will have a more complex proof burden, as we see in Section 4.5.1. With that in mind, we introduce some mechanics to ease the proof of confinement for both λ_{OAST} and for possible future languages.

We separate type checking of base types from security types. This decoupling is easy to achieve in the context of optional security typing. We anticipate that, in a real world system, this should make the creation of useful error messages easier to accomplish. Given we have two layers of type error, errors in base types and errors in information flow labels, it is useful to be able to easily distinguish them terminologically.

We call a term t in λ_{OAST} *well-typed* if there exists a security type S such that $\vdash t : S$. In λ_{OAST}, a *program* is a well-typed term t such that the set of free variables of t is empty, that is $FV(t) = \emptyset$ and $\vdash t : S$: such a term is *closed*. We say a term t is *concrete* if it is well-typed$_c$, that is, every subterm has a concrete label. Proof of

Termination Insensitive Noninterference for λ_{OAST} is only possible over concretely well-typed programs.

Definition 4.6.2 (λ_{OAST} Well-Typedness)**.** A term is *well-typed*$_b$ if it type checks over base types.
A term is *well-typed*$_\star$ if its information flows type check but its typing derivation conclusion has label $\star$.
A term is *well-typed*$_i$ if its information flows type check with its typing derivation having label $\bot$ or $\top$.
A term t is *well-typed*$_c$ if and only if it is *well-typed*$_b$ $\wedge$ *well-typed*$_i$.

The presence of an unknown flow in an λ_{OAST} term 'pollutes' our reasoning about the information flow of its enclosed terms, as we have seen in Section 4.3. This problem can occur at any stage in a partially annotated program. Particularly pernicious are unlabelled sources. In a gradual system, information must be dynamically assigned a label, and any such dynamic assignment is deterministic. While it is certainly possible to enforce *Termination Insensitive Noninterference* via this method, it has the problem of generating *false positives*; flows that ought to have been allowed, but were disallowed due to incorrect inference of the intended security policy. Worse, it might allow information flow to occur which really ought to have been kept secret. This difficulty obliges a developer to add annotations for policy disambiguation; such an obligation runs counter to the intended purpose of gradual typing. Quantified chinks help to address this problem.

4.6.1 Chinks in the Program

The $\star$ label causes problems in our ability to reason about *Termination Insensitive Noninterference*. If a program is well-typed$_\star$ then some of its terms *may* be *chinks*. Informally, a chink is a term in a program that introduces uncertainty about flow

properties. In practice, they appear in two ways in λ_{OAST}: either as a result of partial typing, or deliberately in order declassify information. This last use is the most interesting, and is not yet modelled in any form of gradual security typing known to us. For example, a function of the form $f : \mathrm{Nat}_\star \rightarrow_\star \mathrm{Nat}_\perp$ accepts any integer input, including $\top$. The effect of applying f to a value of label $\top$ (rule SAPP Figure 4.9) is to erase the $\top$ label, and stamp the result as $\star$. We address the lack of safety of this approach with Bayesian information theory estimation, as detailed in Section 4.5.

A chink, therefore, characterises the notion that $\star$ loses or obscures label information. There may be multiple chinks in a program.

Definition 4.6.3 (Chink in λ_{OAST})**.** A *chink* in λ_{OAST} is a lambda abstraction of the form $(\lambda x_\star . y_\ell)_{\ell'}$ such that at least one of ℓ and ℓ' is $\star$ and neither is $\top$.

If the security label on y is $\top$ then the lambda abstraction is not a chink, as the $\top$ dominates $\star$ (Section 4.4.2). Likewise, if the label on the abstraction itself is $\top$, this makes the entire lambda abstraction confidential. Chinks correspond to the locations we want to observe for leakage.

To show that λ_{OAST} confines *Termination Insensitive Noninterference* failures to chinks, we sequentially concretise the $\star$ labels in a program p until we witness a violation of NI within a term (*i.e.* p fails to type check) or we show p is well-typed$_c$ and therefore proves it satisfies NI. For the purposes of proof, we assume that p is a static, "finished" program, in the sense that the only changes to be made are to its security annotations.

Gradualisation of security labels means that p's *security context*, its mapping from program variables to security labels, is in flux. At any point, we have a policy mapping we are trying to achieve as we move from a partially to fully annotated program. This final policy may exist in the mind of the developer only until

the mapping is complete, at which point it can be checked for static correctness. A standard assumption of IFC research is that the policy is fully known to the developer, but there is no reason why this need be the case. Such an assumption falls into the same category as the *competent programmer*. It is a useful abstraction, but often fails when applied in real development. OAST's use of RIF can aid with policy discovery, as it shows flows around typed fragments. When the mapping is complete, we say that the security policy is *concrete* for that program.

It is useful to know how many variables in Γ map to $\star$; this is the $\star$ cardinality of the context.

Definition 4.6.4 (Any Cardinality)**.** The $\star$-*cardinality* of a term t, $|t|_\star$, is the number of $\star$ labels in the free variables in t.

The $\star$-cardinality is a safe over-approximation of the number of chinks; chinks, by definition, must have a $\star$ in their construction. When $|p|_\star = 0$, the mapping is complete: this context is the *policy context*. While the mapping is still incomplete, we call it the *flux context*.

Definition 4.6.5 (Policy Context and Flux Context)**.** The *policy context* of a program p, Γ_p, is a total mapping from program variables to concrete security labels: $\forall v \in FV(p), v \to \ell_c$. The *flux context* of a program p , $\Gamma_\star$, where $|p|_\star > 0$, maps identifiers to labels.

The flux context, a partial mapping of program variables to concrete security labels, needs to stay 'in sync' with the policy context, a complete description of our security policy for our program p. This is naturally described in terms of a relation: we say that the flux context and the policy context of a program p are *policy context related*, $\Gamma_c \approx_p \Gamma_\star$ iff the following condition is met:

$$\Gamma_p \approx_p \Gamma_\star \leftrightarrow \forall x \in \mathrm{dom}(\Gamma_p),\ \Gamma_p(x) = \Gamma_\star(x)\ \vee\ \Gamma_\star(x) = \star \tag{4.3}$$

Policy completion is then the process of turning the flux context into the policy context of a given program. We call this process *label concretisation*.

Definition 4.6.6 (Label Concretisation)**.** Let t be a well-typed term with $|t|_\star \geq 1$. Then a *label concretisation* function $\gamma : (\Gamma_\star, \Gamma_p, x) \to t'$ takes the current flux context, the policy context, and a program variable as arguments and returns a term t' where t' is term t with the label of x concretised as per its value in Γ_p.

In effect, the label concretisation function replicates the ideal developer's knowledge about the security policy of the program. We overload γ to indicate the function that takes a expression and flux context $\Gamma_\star : t$ and sequentially concretises the $\star$ labels in accordance with the policy in Γ_p, which is obviously related to the flux context, $\Gamma_p \approx_p \Gamma_\star$. Then a concretiser of t, $\gamma(t)$, is a function which substitutes instances of $\star$ within a chink until the type of that chink becomes concrete or the chink is no longer typeable. This function is repeatedly applied, monotonically decreasing the $\star$ cardinality, $|t|_\star$, of t. To minimise notation burden, we overload γ to apply directly to labels, *i.e.* $\gamma(\ell_\star) = c$ where c is some concrete label.

To show that a term t in λ_{OAST} with $|t|_\star = 0$ satisfies *Termination Insensitive Noninterference*, we adopt the approach from Zdancewic [223], extended to for our extra base types. The proof is otherwise identical. First, we need to show that the static type system of λ_{OAST}, without $\star$, satisfies preservation. We require canonical forms for proof of progress and preservation, and also for proof of noninterference.

Lemma 4.6.1 (Canonical Forms)**.** *1. If $\vdash v : Bool_\ell$ then $v = True_{\ell'}$ or $v = False_{\ell'}$ and $\ell' \sqsubseteq \ell$*

2. *If* $\vdash v : Nat_\ell$ *then* $v = 0_{\ell'}$ *or* $v = (suc\ n)_{\ell'}$ *and* $\ell' \sqsubseteq \ell$

3. *If* $\vdash v : (S_1 \to S_2)_\ell$ *then* $v = (\lambda x : S_1'.e)_{\ell'}$ *and* $S_1 \leq S_1'$ *and* $\ell' \sqsubseteq \ell$

Lemma 4.6.2 (Progress)**.** *If* $\vdash t :\ S$ *then* t *is either a value* v *or there is some* t' *such that* $t \longrightarrow t'$.

Proof. Standard proof by induction on the derivation $\vdash t : S$ □

Lemma 4.6.3 (Preservation)**.** *If* $\vdash t :\ S$ *and* $t \longrightarrow t'$ *then* $\vdash t' :\ S'$ *and* $S' \leq: S$.

Proof. Standard proof by induction on the derivation $\vdash t : S$. □

The proofs of progress and preservation are entirely standard for this simple type system. In addition to a proof of preservation, we also require that there be a predicate that checks that a closed term t is typeable. Let $TC(t)$ be a predicate that takes a closed term t and provides a proof in the form of a type derivation that t is well-typed. This predicate is the type checking algorithm.

We now have the material in place to prove *Termination Insensitive Noninterference* for λ_{OAST}. Full details of the proof are provided in Section 4.9.1. Formally, the property we want to establish is:

Theorem 4.6.4 (Noninterference)**.** *If* $x : S_\top \vdash e : S_\bot$ *and* $\vdash v_1, v_2 : S_\top$ *then* $e\{v_1/x\} \Downarrow v \iff e\{v_2/x\} \Downarrow v$

Proof. This follows by using the method of logical relations as a special case of Theorem 4.6.5. □

The intuition behind the proof comes from observing the behaviour of a program from the perspective of a low-security observer. If the program is secure, the low-security observer should not be able to see any of the actions performed on sensitive data, while still being able to see the low-security data and computations.

To model this ability to see or not see data, we use an equivalence relation. Given that secure values should be indistinguishable to a low-security observer, we say that two values are related if they cannot be distinguished. If an observer is high-security, then this changes the equivalence relation. Everything should be visible. For this reason, the equivalence relation is parameterised with ζ, the security level of the observer.

The technique of logical relations [136] allows us to extend these equivalence relations to higher-order data and computations. This is identical to that in Zdancewic [223] except that it is extended to also include natural numbers as a base type.

Definition 4.6.7 (Security Logical Relations)**.** For an arbitrary element ζ of the concrete security lattice $\mathscr{L}$, the ζ-level security logical relations are type indexed binary relations on closed terms defined inductively as follows. The notation $v_1 \approx_\zeta v_2 : S$ indicates that v_1 is related to v_2 at type S (*i.e.* indistinguishable). Similarly, the notation $e_1 \approx e_2 : C(S)$ indicates that e_1 and e_2 are related computations that produce values of type S.

$$
\begin{aligned}
v_1 \approx_\zeta v_2 : Bool_\ell &\iff \vdash v_i : Bool_\ell \wedge \ell \sqsubseteq \zeta \Rightarrow v_1 = v_2 \\
v_1 \approx_\zeta v_2 : Nat_\ell &\iff \vdash v_i : Nat_\ell \wedge \ell \sqsubseteq \zeta \Rightarrow v_1 = v_2 \\
v_1 \approx_\zeta v_2 : S_1 \rightarrow_\ell S_2 &\iff \vdash v_i : S_1 \rightarrow_\ell S_2 \wedge \\
&\qquad \ell \sqsubseteq \zeta \Rightarrow \forall v_1' \approx_\zeta v_2' : S_1.(v_1\ v_1') \approx_\zeta (v_2\ v_2') : C(S_2 \sqcup \ell) \\
e_1 \approx_\zeta e_2 : C(S) &\iff e_i : S \wedge e_1 \Downarrow v_1 \wedge e_2 \Downarrow v_2 \wedge v_1 \approx_\zeta v_2 : S
\end{aligned}
$$

To show that a well-typed$_c$ program e produces a ζ-observable output of type S, it is enough to show that $e \approx_\zeta e : C(S)$. From this, it is enough to show

that substitution (*i.e.* the act of evaluation in λ_{OAST}), preserves the η-equivalence relations.

Lemma 4.6.5 (Substitution preserves relations). *If* $\Gamma \vdash e : S$ *and* $\Gamma \vdash \sigma_1 \approx_\zeta \sigma_2 : S$ *then* $\sigma_1(e) \approx_\zeta \sigma_2(e) : C(S)$

Proof. By induction on the typing derivation that a term t has type S. See Section 4.9.1. □

Given that we have fully annotated programs in OAST satisfy *Termination Insensitive Noninterference*, it is a simple matter to show that any leaks cannot occur within fully annotated code, but outside it.

Theorem 4.6.6 (Confinement of *Termination Insensitive Noninterference* in λ_{OAST}). *Suppose t is a closed, well-typed*$_\star$ *program where* $|t|_\star > 0$. *We apply* γ *to a free variable FV(t) in t, giving* $\gamma(t) = t'$, *such that we reduce the number of* $\star$, *then either:*

1. *the new term fails to type check, thereby showing that our program does not satisfy* Termination Insensitive Noninterference *with respect to the policy,*

2. *we complete the concretisation process, demonstrating* Termination Insensitive Noninterference *or*

3. *we reduce the number of chinks where a leak can exist.*

Proof. By induction over the structure of the typing derivation of a term t. See Section 4.9.2. □

$$\frac{\Gamma \vdash t : s \qquad |t|_\star \geq 1 \qquad \gamma(t) = t' \qquad \neg TC(\Gamma' \vdash t')}{\nexists\, s' \,.\, \Gamma' \vdash t' : s'} \;\; (\text{VNI})$$

$$\frac{\Gamma \vdash t : s \qquad |t|_\star = 1 \qquad \gamma(t) = t' \qquad TC(\Gamma' \vdash t')}{\Gamma' \vdash t' : s' \qquad |t'|_\star = 0} \;\; (\text{NI})$$

$$\frac{\Gamma \vdash t : s \qquad |t|_\star \geq 1 \qquad \gamma(t) = t' \qquad TC(\Gamma' \vdash t')}{\Gamma \vdash t' : s' \qquad |t|_\star = |t'|_\star + 1} \;\; (\text{RESTRICT})$$

Figure 4.14: Confinement property inference rules.

The inference rules in Figure 4.14 formalise the description in Theorem 4.6.6. The rule VNI corresponds with point 1 of Theorem 4.6.6, NI with point 2, and RESTRICT with point 3.

Finally, we recall the discussion, in Section 4.5.1, of a simple implementation of OAST in Python. Does this implementation satisfy confinement? Unfortunately, PYTHON's type system does not capture I/O. Moreover, PYTHON does not have an effect-based type system, so global state changes are not expressed in the type signature. Without these features, it is unsafe to assume that the only effects a function have are fully documented in the type signature. A sound embedding of OAST in PYTHON should be possible, especially if a functional programming approach (via the *returns* library for example) were enforced. The security annotation component would also need to be included directly in the type checker, requiring non-PEP extensions to be made. Confinement is easy to prove for the λ_{OAST} language that we discuss in Section 4.4, but proving it for a real production language is considerably more vexed, and will require a large engineering effort.

4.7 Related Work

Sabelfeld and Myers' [165] excellent survey provides a thorough overview of language-based security research as it stood at the turn of the century. Research has been very active in type systems for NI in the years since. It is not possible to cover all the work done in this area, so we focus here on work that bears directly on gradual security typing and QIF.

In recent years, researchers have suggested that gradual typing [176] can alleviate some of the difficulties associated with writing programs having sound information flow. Disney and Flanagan [72] made the first attempt to bring the benefits of gradualisation to the problem of writing information flow secure code when they introduced λ_{gif} in 2011. λ_{gif} is a simply-typed λ-calculus extended with gradual information flow. The base language is similar to our presentation of OAST. λ_{gif} requires explicit security casts in the code in order to enforce NI dynamically. Unlike OAST, the language requires a deterministic treatment of unlabelled values. OAST forgoes this in favour of information theoretic estimation and therefore the type system is not required to make policy decisions. Their language, λ_{gif} treats an unlabelled type as implicitly having the $\top$ label. Casts move the label downwards in the lattice. This is analogous to Thatte's quasi-static typing approach [199]. Quasi-static typing accepts too many erroneous programs statically; it is precisely this issue which triggered the development of gradual typing [176].

Fennell and Thiemann [79] introduce LJGS. This is an extension to a lightweight Java core calculus with a guarantee of secure information flow. LJGS works on the sequential subset of the Java language. They use a static type system supplemented by a dynamic monitor. LJGS also relies on inserting casts into the

source code; such dynamic methods necessarily impact runtime performance. Aware of these problems, they provide a language translation to remove unnecessary casts. Casts are explicit in LJGS; this is a considerable burden as these are not merely annotations, but changes to the source code's logic. OAST only requires annotations, which are entirely optional. The source code of an OAST program does not need to be changed at all, unless the developer learns that it does not satisfy the desired security policy.

A more 'purely gradual' approach comes from Garcia *et al.* [93], who use their "Abstracting Gradual Typing" (Garcia *et al.* [91]) method to derive a gradual source language from Zdancewic's λ_{SEC} [223]. λ_{SEC} itself is a simplified presentation of the SLam calculus of Heintze and Riecke [102]. Garcia and Tanter are concerned with showing that their language $\lambda_{\widetilde{SEC}}$ is a sound extension of λ_{SEC}. Semantics greatly extend those of $\lambda_{\widetilde{SEC}}$. While we do not present, in OAST, a practical industrial language, we plan to scale the technique to meet this task. The chief difficulty is extending the chink identification process to encompass functions that are not purely functional. Toro, Garcia and Tanter extend their work on gradual security typing to consider a language with references [204]. They make the important observation that NI is difficult to reconcile with the dynamic conditions of the gradual guarantee.

The confinement property is related to the blame [212], but is a purely static guarantee intended to ease fault localisation. Optional systems that do not satisfy confinement make reasoning about code unnecessarily difficult. A good example, outside of security, of an optional type system that fails to satisfy confinement is TYPESCRIPT. TYPESCRIPT allows bi-variant subtyping, so even code which is fully annotated cannot be fully trusted, as shown in Listing 4.2 and Listing 4.3.

```
enum EventType { Mouse, Keyboard }
interface Event { timestamp; number; }
interface MountEvent extends Event { p1 : number; p2 : number }
function foo(eventType : EventType; handler:(n : Event) => void) {
    /* ... */
}
listenEvent(EventType.Mouse, (e : any) => console.log(e.p1 + "," +
    e.p2))
```

Listing 4.2: TYPESCRIPT does not satisfy the notion of confinement, here extended to type errors in general, and not just secure information flow.

```
listenEvent(EventType.Mouse, (e : MouseEvent) => console.log(e.p1 +
   "," + e.p2))
```

Listing 4.3: Type checking that does not satisfy confinement.

This code is not type safe[4]. We do not, however, expect type safety in the presence of $\star$. Wishing to show that there are no possible type errors in annotated regions, we concretise the $\star$ type in the final function call.

The resulting code type checks but is still not type safe, rendering the concretisation of the $\star$ as arguably redundant and perhaps even misleading to the developer. Type errors can occur dynamically in fully statically checked code. In an attempt to capture the permissive aspects of the runtime system of JAVASCRIPT, the type system has, perhaps, been made too generous in what it accepts. For security, we definitely wish to avoid undermining confidence in typed regions.

Gradual typing has come under scrutiny for poor performance, following Takikawa *et al.*'s result [194]. Making sound gradual typing efficient enough to

[4] As provided in the TypeScript documentation https://www.typescriptlang.org/docs/

be practical is an active area of research [121] and different solutions have been suggested for different type systems. In practice, industry has largely ignored sound gradual typing in favour of optional typing, an approach which we explore here in OAST. Optional typing does not come with a performance penalty, but does not offer any dynamic soundness guarantees, unless a program contains enough optional types that the type system can detect necessary logical errors and contradictions. While we have instantiated OAST for optional typing, there is nothing to OAST that is necessarily antithetical to gradual typing. RIF can be usefully used with gradual typing as a means for suggesting locations for type annotations.

QIF for estimating leakage has its origins in the work of Clark, Hunt and Malacaria [55]. Various tools for estimating leakage from programs have been suggested. It is not controversial to state that these tools have not made any noticeable impact on industry practice. The QIF Analyser, by Mu *et al.* [137] and QUAIL by Biondi *et al.* [27] compute information flow for simple core imperative languages. Methods for approximating QIF statically exist. Biondi *et al.* [26] use a model-checking based method for quick, approximate measurement of QIF in real world C programs. They use this approach to to quantify the HeartBleed bug. It does not help, however, with the initial *detection* of HeartBleed. One needs to know that the problem is there before trying to quantify it. OAST is useful here as a means to detect problems in an unknown information landscape. ApproxFlow, perhaps more importantly, also requires symbolic execution, with all of its inherent limitations. This assumes the existence of a tool such as CBMC [119] for the target language. It is difficult to scale this approach to testing many different potential chinks. OAST does not require any specialised tools in order to work, though it cannot produce an estimate anywhere nearly as quickly as ApproxFlow.

LeakWatch, by Chothia *et al.* [49] is a leakage quantification tool for Java. LeakWatch uses robust statistical measures to compute a leakage, in bits, from a Java program. This leakage measure comes with a 95% confidence. Achieving this confidence interval requires many tests. LeakWatch does not feature any type enforcement of NI, indeed, it is not a tool which even attempts to enforce a form of security policy. LeakWatch imposes a coding burden on its users: it requires them to insert API calls into their code at each secret point of interest and each observation (public) point. It assumes the user knows where observations need to be placed. LeakWatch's annotation and runtime tax may be high for a program with many secrets and public sinks. The OAST type system tells us exactly where the chinks are and that these are the points in which to perform RIF analysis. We do not need to change the executable code directly.

RIFFLER is a tool for creating information contour maps through a program. This is typically done by testing at least one entry point of a Python program. While RIFFLER is geared towards the analysis of Python, there is nothing in the underlying theory that limits it. We extend RIFFLER by adapting its information theoretic ranking mechanism as detailed in Algorithm 1. RIFFLER has no built-in stopping point for statistically significant estimates of *FlowForward*, nor does it have any ability to enforce security policy. While OAST utilises the convenience of ranked estimates derived from RIFFLER, OAST is the first NI type system to our knowledge that uses information theory to make security labels optional and assess the risk of doing so.

4.8 Conclusion

We have presented OAST, a novel framework for optional security typing. OAST uses information theory to measure the movement of information between se-

curely typed code fragments. In order to reduce the cost of QIF calculations, OAST extends RIF with a principled stopping condition for testing. We reify this with OAST, a pure functional language with optional security typing.

We have also presented a new safety property for optional security languages, namely the *confinement* property. The OAST approach opens doors to new opportunities. Much theoretical and practical work remains to be done. Most importantly, now that the general method has been defined for a core language, and demonstrated the potential in real-world programming, we intend to target an industry programming language with an OAST type system.

4.9 Appendix: Proofs

We present here more detailed proofs of both *Termination Insensitive Noninterference* for OAST, and the confinement property of OAST. As previously discussed, confinement is a natural property of the core OAST language, due to the fact a program readily splits cleanly into subprograms. A proof of confinement for a language such as PYTHON would be considerably more difficult.

4.9.1 Proof of Noninterference

Proof. By induction on the typing derivation that e has type S_c. A well-typed$_c$ term may have sub-expressions which are well-typed$_\star$, however we can prove termination insensitive noninterference for e as long as such sub-expressions are not *chinks*. Considering the last step used in the derivation:

Case (x). Immediate from the facts that substitutions map variables to values and that $\sigma_1(x) \approx_\zeta \sigma_2(x) : \Gamma(x)$ because $\Gamma \vdash \sigma_1 \approx_\zeta \sigma_2$.

Case (bool). $e = b_\ell$. By definition of substitution, $\sigma_1(b_\ell) = \sigma_2(b_\ell)$. By definition, $b_\ell \approx_\zeta b_\ell : Bool_\ell$ as required.

Case (nat). $e = n_\ell$. By definition of substitution, $\sigma_1(n_\ell) = \sigma_2(n_\ell)$. By definition, $n_\ell \approx_\zeta n_\ell : Nat_\ell$ as required.

Case (λ). $e = (\lambda x : S_1.e')_\ell$. Then $S = S_1 \to_\ell S_2$. Assuming that $x \notin \mathrm{dom}(\sigma_1)$, we have $\sigma_1(e) = (\lambda x : s_1.\sigma_1(e'))_\ell$. We need to show

$$(\lambda x : S_1.\sigma_1(e'))_\ell \approx_\zeta (\lambda x : S_1.\sigma_2(e'))_\ell : (S_1 \to S_2)_\ell$$

If $\ell \not\sqsubseteq \zeta$ then the terms are trivially related in s. We assume that $\ell \sqsubseteq \zeta$ and the existence of two values v_1 and v_2 such that $v_1 \approx_\zeta v_2 : S_1$. Now we need to show that

$$((\lambda x : S_1.\sigma_1(e'))_\ell\ v_1) \approx_\zeta ((\lambda x : S_1.\sigma_2(e'))_\ell\ v_2) : C(S_2 \sqcup \ell)$$

By the evaluation rule we have

$$\sigma_1(e')\{v_1/x\} \Downarrow v_1' \wedge \sigma_2(e')\{v_2/x\} \Downarrow v_2' \wedge v_1 \approx_\zeta v_2 : (S_2 \sqcup \ell)$$

By inversion of the typing rule, we have $\Gamma, x : S_1 \vdash e' : S_2$ and that $x \notin \text{dom}(\Gamma)$. It follows therefore that

$$\Gamma, x : S_1 \vdash (\sigma_1\{x \mapsto v_1\}) \approx_\zeta (\sigma_2\{x \mapsto v_2\})$$

By the induction hypothesis, we have

$$(\sigma_1\{x \mapsto v_1\}(e')) \approx_\zeta (\sigma_2\{x \mapsto v_2\}(e')) : C(S_2 \sqcup \ell)$$

which, because $x \notin \text{dom}(\Gamma)$ is equivalent to

$$\sigma_1(e')\{v_1/x\} \Downarrow v_1' \wedge \sigma_2(e')\{v_2/x\} \Downarrow v_2' \wedge v_1 \approx_\zeta v_2 : (S_2 \sqcup \ell)$$

as required.

Case(App). $e = e_1 e_2$ and $S = S' \sqcup \ell$ for some appropriate S' and ℓ. It follows from the well-typedness of e and the induction hypothesis that

$$\sigma_1(e_1) \Downarrow v_{11} \wedge \sigma_2(e_1) \Downarrow v_{12} \wedge v_{11} \approx_\zeta v_{12} : (S_1 \to S')_\ell$$

and

$$\sigma_2(e_2) \Downarrow v_{12} \wedge \sigma_2(e_2) \Downarrow v_{22} \wedge v_{12} \approx_\zeta v_{22} : S'$$

By the definition of $\approx_\zeta$-related values as defined for function types we have

$$(v_{11} v_{12}) \approx_\zeta (v_{21} v_{22}) : C(S' \sqcup \ell)$$

Case(Conditional). $e = \text{if } b \text{ then } e_1 \text{ else } e_2$ where $\Gamma \vdash b : Bool_\ell$ and $\Gamma \vdash e_i : S' \sqcup \ell$. Our goal is to show

$$(\sigma_1(\text{if } b \text{ then } e_1 \text{ else } e_2)) \approx_\zeta (\sigma_2(\text{if } b \text{ then } e_1 \text{ else } e_2)) : C(S' \sqcup \ell)$$

which, by the definition of substitution, is the same as

$$(\text{if } \sigma_1(b) \text{ then } \sigma_1(e_1) \text{ else } \sigma_1(e_2)) \approx_\zeta (\text{if } \sigma_2(b) \text{ then } \sigma_2(e_2) \text{ else } \sigma_2(e_2)) : C(S' \sqcup \ell)$$

If $\ell \not\sqsubseteq \zeta$ then the two terms are trivially related. We assume therefore that $\ell \sqsubseteq \zeta$.

By the induction hypothesis we have that $\sigma_1(b) \approx_\zeta \sigma_2(b) : C(Bool_\ell)$ so by definition we have $\sigma_1(b) \Downarrow v_1$ and $\sigma_2(b) \Downarrow v_2$. Since $\ell \sqsubseteq \zeta$ we have $v_1 = v_2$. Let us assume the condition is true (the false branch is analogous). In this case we have

$$\sigma_1(\text{if } b \text{ then } e_1 \text{ else } e_2) \Downarrow v_{11}, \quad \text{where } \sigma_1(e_1) \Downarrow v_{11}$$
$$\sigma_2(\text{if } b \text{ then } e_1 \text{ else } e_2) \Downarrow v_{11}, \quad \text{where } \sigma_2(e_1) \Downarrow v_{12}$$

By the induction hypothesis we already have $v_{11} \approx_\zeta v_{12} : C(s' \sqcup \ell)$ as required.
Case(BooleanBinaryOp). $e = e_1 \otimes e_2$ and $s = Bool_\ell$ where ℓ is the join of the security levels of e_1 and e_2, $\ell = \ell_1 \sqcup \ell_2$. We have that, for each substitution, $\sigma_i(e_1 \otimes e_2)$ is well-typed and has type s. Therefore, we need to show that

$$\sigma_1(e_1 \otimes e_2) \Downarrow v_1 \wedge \sigma_2(e_1 \otimes e_2) \Downarrow v_2 \wedge v_1 \approx_\zeta v_2 : Bool_\ell$$

From the definition of substitution, we have

$$\sigma_i(e_1 \otimes e_2) = \sigma_1(e_1) \otimes \sigma_i(e_2)$$

By inversion of the typing judgment and two applications of the induction hypothesis, we have

$$\sigma_1(e_1) \approx_\zeta \sigma_2(e_1) : C(Bool_{\ell_1})$$
$$\sigma_1(e_2) \approx_\zeta \sigma_2(e_2) : C(Bool_{\ell_2})$$

Consequently, we have:

$$\sigma_1(e_1) \Downarrow v_{11} \wedge \sigma_2(e_1) \Downarrow v_{21} \wedge v_{11} \approx_\zeta v_{21} : Bool_{\ell_1}$$
$$\sigma_1(e_2) \Downarrow v_{12} \wedge \sigma_2(e_2) \Downarrow v_{22} \wedge v_{12} \approx_\zeta v_{22} : Bool_{\ell_2}$$

By Lemma 4.6.1 we have:

$$v_{11} = (b_{11})_{\ell_{11}} \qquad v_{12} = (b_{12})_{\ell_{12}}$$
$$v_{21} = (b_{21})_{\ell_{21}} \qquad v_{22} = (b_{22})_{\ell_{22}}$$

By two applications of the evaluation rule where have:

$$\sigma_1(e_1 \otimes e_2) \Downarrow (b_{11}[\![\otimes]\!]b_{12})_{(\ell_{11} \sqcup \ell_{12})}$$
$$\sigma_2(e_1 \otimes e_2) \Downarrow (b_{21}[\![\otimes]\!]b_{22})_{(\ell_{21} \sqcup \ell_{22})}$$

Finally, we need to show that

$$(b_{11}[\![\otimes]\!]b_{12})_{(\ell_{11} \sqcup \ell_{12})} \approx_\zeta (b_{21}[\![\otimes]\!]b_{22})_{(\ell_{21} \sqcup \ell_{22})} : Bool_\ell$$

If $\ell \not\sqsubseteq \zeta$ then the expressions are trivially related. Assuming that $\ell \sqsubseteq \zeta$, we need to show that the expressions are equal. If $\ell \sqsubseteq \zeta$ it follows by definition that $\ell_i \sqsubseteq \zeta$ which means, by definition of $\approx_\zeta$ on boolean values, that $b_{11} = b_{21}$ and $b_{12} = b_{22}$, therefore

$$(b_{11}[\![b_{12})_{(\ell_{11} \sqcup \ell_{12})} = (b_{21}[\![\otimes]\!]b_{22})_{(\ell_{21} \sqcup \ell_{22})}$$

as required.
Case(ArithBinaryOp). Analogous to BooleanBinaryOp, but with natural numbers as the base type.
Case(UnaryOp). $e = \neg b$ and $s = Bool_\ell$. By induction on e and the typing derivation of s. □

4.9.2 Confinement Proof

Proof. By induction over the $\star$ cardinality of a program and case analysis over the grammar. We assume that the program **p** has an $\star$ cardinality $|p|_\star \geq 1$. We cover the base cases here. Operations follow the same pattern as APPLICATION.

Case (VARS AND LITS): $\star$ cardinality of a var x is 1. $\gamma(x)$ results in *Termination Insensitive Noninterference* as there is no high-low partition.

Case (ABSTRACTION): let **t** be the term $\lambda x : T.e_\ell$. The $\star$ cardinality of **t** = $|t|_? + |\ell|_? + |e|_? = c$. We now apply γ to concretise 1 security label, creating a new term t'. If $\neg TC(t')$ then we have VNI. Otherwise we reason over the size of c; if $c = 1$ and $TC(t')$ then we have NI, otherwise we have shown that there was no leak present and rule RESTRICT applies.

Case (APPLICATION): let t be a well-typed term of the form $e \cdot e_1$. The $\star$ cardinality of t is $|e|_? + |e_1|_? = c$. If $|e|_? \geq 1$ then we apply γ to e, otherwise we perform $\gamma(e_1)$. By definition, $|t|_? \geq 1$. Let $t' = \gamma(t)$. If $\neg TC(t')$ then we have VNI. Otherwise we reason over the size of c. If $c = 1$ and $TC(t')$ then t' is a concrete term and we have NI, otherwise we apply the RESTRICT rule. □

Note that the RESTRICT rule is essentially our induction hypothesis. We do not need to complete the inductive process to show the confinement property, but by following the logic to its conclusion and monotonically applying γ, we always arrive at one of the two base cases, either VNI or NI.

Chapter 5

SafeStrings

The versatility of strings makes them a powerful extensibility mechanism; they enable universal `varargs` at the price of a bit of parsing. Internet APIs, such as RESTful APIs, make extensive use of strings for precisely this reason. Such expressive power comes at a cost; it is easy to introduce bugs via malformed strings or incorrect string handling. The latent structure within strings is opaque to most type checkers. So far, attempts to provide safe string-based extensibility provide insufficient safety or lose extensibility. We introduce SAFESTRINGS to square the circle and provide safe extensibility for data-carrying strings. SAFESTRINGS harness latent structure and expose it to the type checker. SAFESTRINGS use the subtyping and inheritance mechanics of their host language to create a natural hierarchy of string subtypes. This makes them more expressive than *regex*-validated strings. A drop-in replacement for strings, SAFESTRINGS are less rigid than DSLs, preserving the powerful extensibility of strings. SAFESTRINGS are lightweight, deployable, and optional: we demonstrate this by both retrofitting two programming languages, Java and TYPESCRIPT, with SAFESTRINGS and also show their inclusion from the ground up in BOSQUE, a new language with native support for

SafeStrings.

5.1 Introduction

Strings are ubiquitous in code; their versatility makes them an flexible encoding mechanism, and a universal means of moving information. It is no wonder that RESTful APIs [81] make extensive use of them. Strings allow a developer to interact with and within systems without constantly creating new types; they provide easy extensibility. This extensibility can be crucial to a project's viability when a system's requirements have not gelled.

Great utility and flexibility also means strings can easily introduce bugs [73]. Unsurprisingly, most programming communities discuss string validation at length, especially those that target string heavy problem domains. Discussion of *string literals* [161] and *regex*-validated strings [205], in TypeScript and other languages, are just two examples. String handling bugs often arise due to the *latent structure* hidden within many strings. This latent structure, all mono-typed under the umbrella 'string', is a implicit, type system embedded within the parent type system. The type checker has limited power to help the developer avoid mistakes. Refactoring strings to appropriate data types is laborious and comes at the cost of making the system harder to extend.

We introduce SafeStrings to preserve the extensibility of data-carrying strings while providing type safety guarantees. SafeStrings expose a data-carrying string's latent structure to a type checker, so that the checker can ensure the structure's integrity. As SafeStrings use annotations only, it is easy to revert code to purely typed. This is not the case when producing custom types for every string. SafeStrings rest on the observation that data-carrying strings are

Abstract Data Types (ADT). SAFESTRINGS are a principled way to embed the data into the string ADT of the language, improving both safety and the flexibility of string. Notable examples of such ADTs are csv strings, lists, and records to represent postal addresses. This principled embedding means a developer can treat a SAFESTRING solely as a *string*, and not as some more complex object, while enjoying two assurances of type checking: that a SAFESTRING contains only valid data and assigning an instance of one SAFESTRING into another incompatible type triggers a type error.

It is easy to write new SAFESTRINGS (Section 5.4.3), making it convenient to move the intrinsic complexity of dealing with latent string structure away from front-end developers to library authors. SAFESTRINGS are string subtypes, so one can always pass an appropriate subtype without having to rewrite code. SAFESTRINGS can define a subtype relation over structured strings more easily than via language inclusion (Section 5.3.2). They permit 'type-safe' string manipulation, in the sense that the result of an operation is always well-typed under a SAFESTRING, as long as its grammar accepts the serialisation of the result (Section 5.3.4). String validation becomes membership checking and occurs when converting a raw string to a SAFESTRING; revalidation is either unnecessary or requires only rechecking substrings for membership.

Fortunately, most uses of data-carrying strings reuse a small set of structures: our empirical study of a Java corpus found that just 10 structures covered 100% of all data-carrying strings in a uniformly at random sampling of the corpus (Section 5.5.2). This means that developers often need only replace their language's raw string with the appropriate SAFESTRING annotation from our SAFESTRING, as our case studies illustrate (Section 5.5). Annotations can be added incrementally, or removed. As the underlying program type checks with , remov-

ing the annotation simply reverts the type to . This permits easy incremental type migration [220].

The requirements for SAFESTRINGS are common in modern languages: user-defined data types, subtyping, and static type checking. This means SAFESTRINGS are essentially language agnostic and applicable to any object-orientated language. When using a language *retrofitted* with SAFESTRINGS (Section 5.4), a developer need only annotate a string variable to declare its SAFESTRING type. Because SAFESTRINGS store and reuse the underlying string structure while still maintaining the string interface, they are more compact and efficient than *PCRE2*[1] validated strings [152], which are a verbose encoding and just perform a syntactic validity check. *PCRE2*, in general, is known to be difficult to read, write and maintain [135]. Moreover, by discarding structural knowledge, *PCRE2*-validated types do not allow for easy subtyping, nor method specialisation (Section 5.6).

Domain-specific languages (DSL) extend their host language to ease reasoning about and solving domain-specific problems; DSLs focus on code, not data. Their goal is to expose latent semantics, not latent structure. DSL code regions necessarily have different semantics than their host language. When a DSL handles latent string structure, exposing that structure to its host language is usually challenging. For example, a syntax-embedded string is no longer a drop-in replacement for string. In contrast, SAFESTRINGS focus on data, not code; they do not change or extend their host language's semantics, but leverage its existing abilities. SAFESTRINGS expose latent string structure to their host's type checker. This focus gives SAFESTRINGS their power and concision, making them readily deployable. SAFESTRINGS aims at the 'sweet spot' between deployable,

[1] *PCRE2*, "'Perl-compatible regular expressions", are often called *regex* in industry, despite being much more powerful than regular expressions.

but brittle *PCRE2*, and a bespoke DSL with its complex interactions with its host language (Section 5.6).

To evaluate SAFESTRINGS, we first retrofit TYPESCRIPT and Java, thus showing the language agnosticism of SAFESTRINGS. By using Java, we demonstrate SAFESTRINGS for a nominal type system, and in Section 5.3.2 for a structural type system, such as TYPESCRIPT. We describe the construction of a library of SAFESTRING definitions and demonstrate a possible concrete syntax for both TYPESCRIPT and Java, which we implement via preprocessing (Section 5.4.1). We show a language, BOSQUE [129, 128], where SAFESTRINGS are native, first class citizens and demonstrate their usage (Section 5.4.4). We discuss several case studies of real code (Section 5.5), including filepaths, *css* hex colour strings, and *css* units. Leveraging the core library, we applied SAFESTRING annotations to a corpus of Java programs to measure its annotation burden on existing projects (Section 5.5.2). We also measure the performance impact of our library in terms of SAFESTRING serialisation and deserialisation costs.

Our principle contributions are:

- To address the latent structure problem for strings, we introduce SAFESTRINGS, a lightweight, language-agnostic, and seamlessly deployable programming model for string type checking (Section 5.3);

- We show how SAFESTRINGS facilitate the definition and verification of a subtype relation under strings (Section 5.3.2) and type safe operations over strings (Section 5.3.4); and

- We present realisations of SAFESTRINGS in Java, TYPESCRIPT and BOSQUE (Section 5.4).

5.2 Motivating Example

Bugs from mishandling strings in this manner are common[73]. An analysis of primitive type usage in SF110 [85] shows that strings are commonly used in complex control points. Given that strings are a common source of bugs, and strings are used in complex ways, handling strings at the type level can obviate a class of bugs. Of nine basic types studied in [85], strings account for approximately 22% of control points in all classes. Only booleans and integers are more used. Of these 9 basic types, 6 are different representations of number. No such recognition of plurality is afforded to string: a reasonable corollary is that the uni-typing of strings leads to errors.

Apache bug OFBIZ-4237[2] results in a execution ending exception being thrown if a string parameter does not contain a colon, ":" (Listing 5.1). The code takes in a string from a call to `readLine` (in a method `processClientRequest()`) which is sent for processing by the `private` method `processClient()`. This method returns a `String`. The returned `String` takes the form of a flexible enumeration, a common use for strings as it makes for a convenient varargs mechanism. It is difficult to envisage how to remove `String` entirely from this code without a massive refactoring effort touching other parts of the project. This is due to the `readLine` call, to matching a substring in the `request` to a string in a config file, and finally to the the requirement for a flexible, meaningful, return value.

Listing 5.1 was patched with an additional check for the presence of the colon delimiter. The patch itself is simple, but this code passed through all quality control mechanisms and resulted in an error caught by the end user. The error is more easily catchable by the addition of a SAFESTRING annotation for a colon-

[2] `https://issues.apache.org/jira/browse/OFBIZ-4237`

```
String request = reader.readLine();
if (request != null) {
  writer = new PrintWriter(client.getOutputStream(), true);
  String key = request.substring(0, request.indexOf(':'));
                                      ▲ /* unsafe string handling */
  String command = request.substring(request.indexOf(':') + 1);
  if (key.equals(config.adminKey)) { ··· }
```

Listing 5.1: Line 1 takes a string from an outside source. This string is split on a colon delimiter in line 4, but the string does not guarantee a colon is present. This leads to an error that can be difficult to spot.

delimited string, as in Listing 5.2.

The annotation in Listing 5.2 introduces a dynamic check, much in the manner of gradual typing[174]. However, unlike with gradual typing, it is now easy to simplify the code as an optional refactoring exercise Listing 5.3. Such simplification allows the cost of dynamic SAFESTRING instantiation to be offset against corresponding code reduction.

With SAFESTRINGS is it easy to model partially structured strings. The client request of listing 5.2 returns more than just a colon delimited string. It returns a string structured around that colon. The substrings surrounding the colon may have additional structure. This is easy to model with SAFESTRINGS, as we can use other SAFESTRINGS to check and type the available substrings, *i.e.* we can have the type @Colon(@Comma)(@String), where we have a hierarchy of concerns and structural obligations. Due to SAFESTRINGS using native subtyping and objects, this is easy to achieve. We can use the @String annotation to allow tagging of data.

The real power of SAFESTRINGS comes in the sequences of method calls. If Listing 5.3 returns the colon delimited string as its result, then all meth-

```
        ▼ /* only a Java annotation required */
String @ColonDelimitedSafeString request = reader.readLine();
if (request != null) {
  writer = new PrintWriter(client.getOutputStream(), true);
  String key = request.substring(0, request.indexOf(':'));
  String command = request.substring(request.indexOf(':') + 1);
    ▲ /* both key and command could benefit from annotations */
  if (key.equals(config.adminKey)) { ··· }
```

Listing 5.2: The addition of a simple SAFESTRING type annotation inserts a dynamic type check that `request` contains a colon. Any annotated methods that use this string can be statically assumed to have the correct structure. Adding annotations to both `key` and `command` will provide even more string safety.

ods calling it can use the information provided via the annotation for purely static type checking. If we wrap the code fragment Listing 5.3 in a function `String @ColonDelimitedString foo()`, and given a method `void bar(String @CommaDelimitedString s)`, then we can now statically reject `bar(foo())`. The runtime cost of a SAFESTRING is only paid once, when the string is instantiated, so over the course a program run, the cost is amortisable.

5.3 SafeStrings

SAFESTRINGS are abstract data type for strings. We make them via such familiar tools as objects, grammars and parsers. First, we present Definition 5.3.1 in Section 5.3.1. When working with SAFESTRINGS, you want to know whether an operation on, say, an email string produces an email string as output: we discuss this closure problem when discussing the definition. Essential to the SAFESTRING programming model is that SAFESTRINGS should be indistinguish-

```
@ColonDelimitedSafeString request = reader.readLine();
if (request[0].equals(config.adminKey)) { ··· }
```

Listing 5.3: From 6 lines to 2: using a *ColonDelimitedSafeString* drastically simplifies the code. The developer no longer needs to write checks each time, but rely instead on one dynamic assertion and static reasoning otherwise. If `request` passes the check, then the field access in line 2 is certain to succeed.

```
String @ColonDelimitedSafeString(@Comma)(@String) request = reader.
    readLine();
/* request := ``csv,list,of,data:metadata which we want to store'' */
request.append(``safe'');
/* request := ``csv,list,of,data,safe:metadata which we want to store
    '' */
```

Listing 5.4: We can use SAFESTRINGS to carry metadata about the string that does not interfere with type safe methods, but is still available for reading and writing.

able from strings from the user perspective: this requires method overriding, which we discuss in Section 5.3.4.

5.3.1 Definition

Informally, SAFESTRINGS combine a string, a grammar, a parser and a structured, internal representation (Definition 5.3.1). This combination creates an algebraic data type (ADT) to make implicit structure explicit for better string type checking. To instantiate a SAFESTRING, a parser first checks a string for language membership. The SAFESTRING stores the parser output in its internal representation, which makes explicit the SAFESTRING's structure. The user never manipulates this internal representation directly, but rather an appropriate serialisation of it.

This serialisation comes via a `cast()` function and must satisfy an equivalence law (Equation (5.1)). A SAFESTRING represents the set of all strings that its grammar accepts. We now define SafeStrings.

Definition 5.3.1 (SAFESTRING)**.** For a host language H with the string type S_H, a SAFESTRING defines a subtype of S_H via the 4-tuple $\langle G, \rho, \phi, \alpha \rangle$, with the following types

$G : (N, \Sigma, P, S)$	grammar
$\rho : AST^H$	a structured object for storing the AST
$\phi : \mathrm{S}_H \rightharpoonup AST^H$	parser
$\alpha : AST^H \to \mathrm{S}_H$	serialiser

A SAFESTRING's parser ϕ parses the unambiguous context-free grammar G [184]. We limit ourselves to languages recognisable by CFGs because we are interested in capturing those strings that encode *data*, rather than *code*: common structured strings, such as filepaths and delimited strings (Section 5.4), are all CFG recognisable. We require $\Sigma \subseteq \Sigma_H$.

Instances of a SAFESTRING range over $L(G)$, all those strings ϕ parses. When it succeeds, ϕ outputs an instance of AST ^{H}, an abstract type that encompasses all ways to encode structure in H, which a SAFESTRING stores in ρ. Conceptually, ρ stores and makes *explicit* its raw string's structure, which would otherwise be submerged in ST_H. ρ combines a record and metadata, storing an instance of AST^H. α uses the metadata to recover ST_H's raw string, modulo $\approx_t$ (See Equation (5.1) below). A SAFESTRING calls ϕ to initialise ρ in a constructor; when s is the raw string passed into the constructor, $\rho = \phi(s)$, if ϕ succeeds. The parser ϕ is partial to capture the possibility of failure. The handling of the partiality of ϕ is H specific. A parse error generates a type error: the input string is not in

$L(G)$ and therefore does not have the same type as the SAFESTRING declaration. Listing 5.11 shows how to make error messages more informative. It is beneficial for method overriding (Section 5.3.4), which SAFESTRINGS use to seamlessly integrate into S_H, and the space and time efficiency of SAFESTRING operations if ϕ is compositional, *i.e.* decomposes into subparsers that can separately handle fields in ρ (Section 5.3.2).

We use α to recover the string, s, from ρ. Conceptually, it is akin to a pretty printer. It is insufficient, however, for α to be merely a literal serialisation of ρ. For interoperability with ST_H, ϕ and α of the SAFESTRING ß must, when ϕ is defined on s, satisfy the following equivalence

$$s \approx_t (\alpha \circ \phi)s \tag{5.1}$$

where $s \in \Sigma^*$. The *token equivalence relation*, denoted $\approx_t$[3], does not require equality over non-tokenisable characters, *e.g.* `"foo "` $\approx_t$ `"foo"` *iff* trailing whitespace is ignored during tokenisation. The equivalence defined in Equation (5.1) is the algebraic specification of the SAFESTRING ADT [74]. This equivalence can cause some problems with equality, as the SAFESTRING version of `foo` may not be equal to the original input string. We discuss this further in Section 5.3.4. In general, we refer to α as the `cast()` method in code fragments, though we reify `cast()` via familiar `toString()` methods in our implementations.

The fundamental insight for SAFESTRINGS rests on Equation (5.1). By building a transformation between string and structure mediated by Equation (5.1), it is possible to have the best of both worlds: a simple string programming model, with the power of an object model. Previous work on typed strings does not have ρ, therefore it performs a membership check but does not convert and store the

[3] Not to be confused with the OAST type consistency relation of Figure 4.8

AST (Section 5.6). The hard work of verification is discarded, rather than kept and utilised. Alternatively, other methods translate the string into a record type and leave it as such. The input is no longer a string and cannot be used conceptually as a string. The programming model is altered: SAFESTRINGS address this problem.

Grammars, their associated parsers, and pretty printers can be huge. Despite this, SAFESTRINGS are, in practice, quite small. This is due to their focus on encoding structured data. Using SAFESTRINGS requires adding new type annotations to code. Here too, the reuse of implicit structure in code keeps low both the cost of defining SAFESTRINGS and cost of annotating strings to use them. Section 5.5.2 quantifies these low costs over our Java corpus.

SAFESTRINGS rely on the host language's type checker for type safety. This is both a positive and negative feature. SAFESTRINGS cannot guarantee type safety beyond the guarantees of the host language: thus, the static power of typed strings differs per language. TYPESCRIPT allows bi-variant subtyping by default, which undermines subtyping guarantees. Other languages, such as Java and BOSQUE, do not permit this type of error. Adding SAFESTRINGS to an existing language compliments the existing type safety guarantees.

Closure. When operating on a SAFESTRING instance, such as an email address, you may want to know that an operation produces a valid email address or errors. Operations over strings are much simpler: functions over strings, S^*, of the form $f : S^* \to S^*$ are endomorphisms: such endomorphisms do not require dynamic checking. This is not true, in general, for SAFESTRINGS. Many standard string operations, such as concatenation, when lifted to a SAFESTRING, σ, are no longer automatically endomorphisms in $L(\sigma)$, unlike with S_H. They require language membership checks. Other common methods, such as `replaceAll()`, change characters that might alter the SAFESTRING type of a string; *i.e.* changing

the comma to semicolon in a *CommaDelimitedString* means the result is still a delimited string, but not a *CommaDelimitedString*, whereas changing the delimiter on the super-type `DelimitedString` does not have this effect. It is difficult to know statically what actions over a SAFESTRING are necessarily endomorphisms. We handle this problem in a practical fashion by *lifting* all SAFESTRING types. A type is lifted *iff* it has bottom, $\perp$, as an element. In Java and TYPESCRIPT, such lifting is automatic, as every type has access to exception handling mechanisms, but this is not necessarily the case.

5.3.2 The SAFESTRING Subtype Relation

We may categorise subtyping broadly between nominal and structural: informally, nominal subtyping has it that one type is a subtype of another if so declared (*i.e.* a naming relationship is sufficient), whereas structural subtyping holds that two types have a subtype relation if they share corresponding elements, regardless of any declared relationship. In languages with structural subtyping, the easiest way to induce a subtype relation is to include the sub-parsers as fields in the record (Definition 5.3.2). Structural subtyping is guaranteed then by adding extra parsing fields and adjusting ϕ accordingly.

SAFESTRINGS use ρ to produce a subtype relation for strings (Figure 5.1). The soundness of this subtype relation is controlled by the parsers, which act as filters over ST_H. To make the presentation programming language agnostic, we discuss subtyping in terms of records. Exact implementation is language dependent. The subtype relation for SAFESTRINGS is most easily presented in terms of width subtyping [154]. Informally, width subtyping incorporates additional fields into a record, making it 'more informative' than its super-type. Because SAFESTRINGS have ρ, it makes it much easier to define a subtype relation in terms of objects with fields, rather than via language inclusion (Section 5.6).

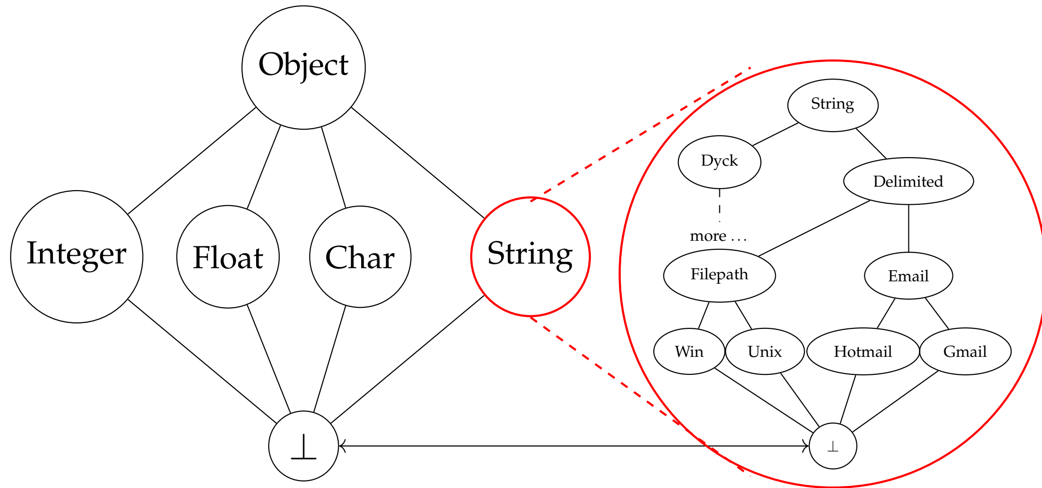

Figure 5.1: Given a subtype hierarchy, SAFESTRINGS create a local subtype hierarchy within `string`. The error state, $\perp$, is shared between SAFESTRINGS and the "external" hierarchy. $\perp$ occurs when a string fails to satisfy the requirements for a particular SAFESTRING. All the SAFESTRING types, apart from $\perp$, can be assigned the type `string`.

We have implemented SAFESTRINGS for both structural and nominal type systems (Section 5.4). For the nominal type system, the subtype relation is easy to enforce. For the structural type system of TYPESCRIPT, we chose to increase width via parser combinators; these provided us with the necessary extra fields.

Definition 5.3.2. SAFESTRING Subtyping.
Let ρ be a record with named fields $\rho = \{x, ..., z\}$; ρ has type τ. Each field has an associated parser, $\{x_p, ..., z_p\}$, also part of ρ. Then a *subtype* τ' of τ is the collection of named fields $\{x, ..., z\}$ and an associated collection of parsers $\{x_p, y_p, ..., z_p\}$.

where y_p is a new parser not in $x_p, ..., z_p$. This definition is independent of the correctness of the parsers *w.r.t* the language they are intended to recognise: subtyping still works for incorrect parsers during type checking level; the problem

being the subtype relation is not the one intended by a developer.

Of course, simply adding extra sub-parsers is not very principled. We require that both y_p and some parser y'_p already in the parsing collection both examine the same part of the input string, one acting as a specialisation of the other. Two degenerate cases can occur: when $y_p = y'_p$ and when the intersection of the languages recognised by y_p and y'_p is empty, *i.e.* $L(y_p) \cap L(y'_p) = \emptyset$. We leave it to the developer of a SAFESTRING definition to ensure that this does not happen.

5.3.3 Equivalence of Operations over SAFESTRINGS and Strings

Definition 5.3.1 says nothing about operations on SAFESTRINGS. In practice, all OOP languages define a string interface, a collection of methods available over all strings. To maintain the string-like quality of SAFESTRINGS, we must expose at least the same interface. We also consider that a SAFESTRING might add, or override, methods. This is for convenience and expressiveness. We do not want to permit adding or overriding methods that violate the string contract of the host language interface: it would be contrary to the string API in Java, for example, if we were to add an `eval()` method as integral element to a SAFESTRING. To ensure that no added or overridden methods fundamentally alter the string interface, we define a limit on what may be added (Definition Definition 5.3.3). The easiest way to implement a sane string interface for a SAFESTRING is to copy the native string interface over to the parent SAFESTRING object: this is the core of the method we use for both Java and TYPESCRIPT.

At a minimum, a SAFESTRING must implement the entire set of methods of the host language string interface, S_{adt}. We call this implementation Safe_{adt}. To add or override a method, m, to Safe_{adt}, we stipulate that the result of calling m must be reproducible via a finite number of methods calls in S_{adt}. If we add a method, for example, to an email SAFESTRING that extracts the domain element, `String`

`getDomain()`, then the same functionality is replicable via substring searching and splitting over a raw string that encodes an email address.

Definition 5.3.3. Sound Interface Extension
A method m_{ex} is a sound interface extension to Safe$_{adt}$ iff there exists a finite sequence of methods $\sigma = m_0 \cdots m_x \in \mathrm{S}_{adt}$ such that, for a string s, $m_{ex}(s) \approx_t \sigma(s)$.

For a SAFESTRING implementation to be a safe drop-in replacement for string, it needs to sound *w.r.t* the string interface of H.

Theorem 5.3.1. SAFESTRING *Soundness*
Given a Safestring type S_{st}, and for all strings, s, such that $S_{st}(s)$ exists, and for any valid sequence of operations, $m = op_0 \cdots op_x(s)$, in S_{st}, there exists a finite sequence of operations in S_{adt} $\sigma = op'_0 \cdots op'_x(s)$, such that $m_{st}(s) \approx_t \sigma(s)$.

By valid, we mean that the sequence of operations type checks in the host language.

Proof. The proof is per SAFESTRING definition, and proceeds by induction over the length of σ. The base case requires that Equation (5.1) holds for each individual method of Safe$_{adt}$. $\square$

Equality. Equality presents no problems when comparing two SAFESTRINGS of the same type. More delicate, however, is comparing a string against a SAFESTRING for equality. Consider a program, p, that takes in a user input string s (Figure 5.2). This string is a filepath that contains trailing whitespace. At some point in the program flow, s is instantiated as a *FilePath* SAFESTRING. This effectively strips any trailing whitespace, as whitespace at the end of a filepath is not tokenisable. Equality between the original string, s and the SAFESTRING version s' no longer holds. To address this, when comparing a string against a

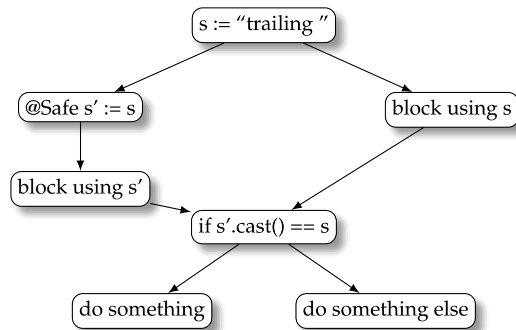

Figure 5.2: A string might fail to equal itself if equality is handled naively. s' is a SAFESTRING which has had its whitespace automatically trimmed. As a result, it is no longer equal with "itself", the original input string, s.

SAFESTRING, we attempt to coerce the string to the appropriate SAFESTRING. If this fails, then **false** is returned.

5.3.4 Operation Overriding

A SAFESTRING includes an internal representation. Operations over these representations are often simpler to perform and reason about than operations over the raw string. One need only touch individual fields or sequence elements, without reprocessing the entire string. An operation such as `replaceAll()` can often be viewed as 'tree surgery': changing all delimiters in a delimited SAFESTRING via `replaceAll()` is just changing the contents of the parent node, from ',' to ':', for example. Unfortunately, not all operations are as simple as this. SAFESTRINGS open the prospect of increasingly type-safe string operations beyond `replaceAll()` and similar functions. We have already seen this idea in Section 5.1 in the discussion of `cssColour` strings, an example where it is necessary to operate over the entire AST, but doing so results in a rational, type safe function (Listing 5.5).

```
/* @param{return : cssColour} */ concat('#001', '#100') => '#101'
concat('#001', '#100') => '#001#100' /* probably not what we want */
```

```
/* the specialised concat function. This is more efficient, and
   statically type
safe, if c1 and c2 have already been annotated as cssColour strings
   */
concat'(c1 : string, c2 : string) : cssColour {
  let c1_ = new cssColour(c1);
  let c2_ = new cssColour(c2);
  let ret = new cssColour(cssColour.add(c1.rgb, c2.rgb).toString());
  return ret;
}
```

Listing 5.5: Concatenating two `cssColour` SAFESTRINGS together in TYPESCRIPT is guaranteed to produce a well-formed `cssColour` SAFESTRING. This operation is closed over its type due to the fact that a `cssColour` SAFESTRING is structurally a natural number: `cssColour` concatenation operation becomes (overflow-aware) addition over natural numbers. The annotation in the doc string enables to choice of an appropriate definition of `concat`. When the input strings are not typed, we need to attempt to coerce the arguments to SAFESTRINGS.

Another common operation over a string is slicing, *i.e.* extracting a substring. Slicing in SAFESTRINGS rests on the intuition that we normally look to extract a particular substring from a string, and not just a random slice. These particular substrings are likely to be represented a SAFESTRING's internal structure ρ as individual elements or the composition of individual elements. Slicing becomes projection from the internal representation. Listing 5.6 shows one way to perform this in TYPESCRIPT. Given a *regex*-validated string, one still needs to write `email.split('@')[0]`. With a SAFESTRING, one simply extracts the named field, *i.e.* `email.name`, an action that has no additional runtime cost. As an `email.name`, its

character set is statically known, and its subparser is still dynamically available. Moreover, this form of slicing cannot fail silently, as in the second example in Listing 5.6. A user desiring free slicing over a SAFESTRING need only manipulate the raw strings as opposed to its representation, *i.e.* take a slice from the `cast()` of the SAFESTRING.

```
"name@email.com".split('@')[0] // OK: returns 'name'
"nameemail.com".split('@')[0] // Bad: returns 'nameemail.com' without
    warning
```

Listing 5.6: Slicing in TYPESCRIPT can fail silently and is easy to get wrong. SAFESTRINGS make many common slicing operations simpler as the internal representation is the original string "sliced" into syntax elements.

5.3.5 Locally Gradual Typing

There is flexibility for the user in using SAFESTRINGS and strings: casting a SAFESTRING to `string` allows a program to statically type check when otherwise it might not. We call this flexibility *locally gradual typing* (*LGT*).

Gradual typing is essentially a typing discipline allowing a free mixture of static and dynamic type checking. Different flavours exist: *LGT* most closely resembles quasi-static typing [199]. Quasi-static typing is a combination of partial types and the automatic insertion of implicit positive and negative coercion. This accurately describes the action of the program transformation (subsection 5.4.1). A quasi-static system divides programs into three categories: well typed, ill typed and ambivalent. *LGT* relies on subtyping and treats the $\star$ type as the top of the subtype hierarchy (Figure 5.1). Siek *et al.* [177] show that the general form of quasi-static typing suffers from the fact that implicit downcasts, combined with the transitivity of subtyping, creates a fundamental problem that allows ill typed programs to exist, even when all parameters are annotated.

SAFESTRINGS utilise the intuition that the problem of quasi-static typing is much simpler when it is local: rather than $\top$ being the absolute top in the type hierarchy (*e.g.* Object in Java), let $\top$ be `string`, and the subtype hierarchy the lower set of `string` whose elements are strings parameterised by language membership. More formally, given the subtype hierarchy, O, of a programming language, the quasi-static lower set, $S^{\downarrow}$, is $\forall s \in S^{\downarrow}, \forall y \in O, y \leq S^{\downarrow} \Rightarrow y \in S^{\downarrow}$, where we set $O_{\top}$ to be `Object` and $S_{\top}$ to be `string`. Given this, the danger of casting to and from absolute top $\top \in O$ is removed. Casting to $\top \in S^{\downarrow}$, notated $\top_{s^{\downarrow}}$, is safe, because all elements below $\top_{s^{\downarrow}}$ are `string` filtered by language membership, or an exception. We cannot transfer out of the `string` element of the type hierarchy, unlike with standard quasi-static typing (Figure 5.1).

The partial (string) types are defined by:

$$\tau ::= g_i \ldots g_j \mid \top_s | \tau \rightarrow \tau$$

where $g_i \ldots g_j$ are typed strings parameterised by language membership, such that $g_i \neq g_j$, when $i \neq j$. The type $\top$ is assignable to all objects g. $\top$ is not assignable to the special object $\perp$, which signifies a runtime error, and function types.

In Figure 5.3 and Figure 5.4 we see the three categories of locally gradual program: well typed, ill typed, and ambivalent. Different degrees of annotation provide different degrees of guarantee. When there are no annotations at all (Figure 5.3), the program type checks but fails dynamically. When annotations are complete, as in the right-hand examples of Figure 5.4, then the resulting code is statically type safe *w.r.t* string usage. The other two examples have partial annotations and show the ambivalent aspect of locally gradual typing for strings. Only if `foo` is annotated (the first method to be called), can good static guarantees be made, otherwise SAFESTRINGS capture that subset of errors which is within scope of the annotations. The partially annotated code, if statically acceptable,

```
function sep(ipv4 : string) : string { return ipv4.split(``.''); }
function fetch(url : string) : string { return get_ipv6(url); }
console.log(sep(fetch(api_input())));
```

Listing 5.7: Well-typed string manipulating code that has two sources of error. Even if the API input is a well-formed URL, there is still a problem with IP versions.

```
function sep(ipv4 : string) : string { return ipv4.split(``.''); }
function fetch(url : string) : IPv4SafeString { return get_ipv4(
    url); }
console.log(sep(fetch(user_input())));
```

Listing 5.8: Partial type annotations reduce the scope for errors. The programmer has fixed the IP version problem via SAFESTRING type annotations, but API input has still not been validated.

```
function sep(ipv4 : IPv4SafeString) : string { return ipv4.split
    (``.''); }
function fetch(url : URLSafeString) : IPv4SafeString { return
    get_ipv4(url); }
console.log(sep(fetch(user_input())));
```

Listing 5.9: SAFESTRINGS prevent the errors of the previous code fragments. If API input is malformed, then the developer gets a type error, which helps debugging.

Figure 5.3: Different stages of SAFESTRING type annotations. The code on the left type checks, as everything gets and returns a string, but produces a dynamically generated error via an exception mechanism. The code on the right is partially SAFESTRING annotated. This type checks, but fails with a runtime error.

Partial	Complete

Partial

```
/* CommaDelim : \safe that guarantees the presence of a comma */

/* @param{s1 : string}
   @return{CommaDelim} */
function foo(s1 : string) : string {
    return s1;
}

function boom(s2 : string) : string {
  return s2;
}

function baz(s3 : string) : string {
    return s3.cast();
  }

/* type checks */
let good_test = (baz(boom(``this.is.a.test'')));
/* does not type check */
let bad_test = (baz(boom(foo(``this.is.a.test''))));
```

Complete

```
/* CommaDelim : \safe that guarantees the presence of a comma */

/* @param{s1 : CommaDelim}
   @return{CommaDelim} */
function foo(s1 : string) : string {
    return s1;
}

/* @param{s2 : CommaDelim}
   @return{CommaDelim}
*/
function boom(s2 : string) : string {
  return s2;
}

/* @param{s3 : CommaDelim} */
function baz(s3 : string) : string {
    return s3.cast();
  }
/* does not type check */
let bad_test = (baz(boom(foo(``this.is.a.test''))));
```

Figure 5.4: The code on the left is partially annotated. Compilation fails with a type error only when calling `foo`. When `foo` is not called, the code runs as expected. The code on the right is fully annotated. `bad_test` does not type check. Due to complete annotations, the code itself requires fewer manual checks.

has the same semantics as the unannotated code.

LGT with Gradual Typing. When $\top$ is ambiguous, *i.e.* not statically determined to be `string`, then *LGT* may suffer from the same problems as quasi-static typing. Such a situation occurs in languages with gradual or optional type systems, such as TYPESCRIPT. If $\subseteq$ is the subsumption relation in an optionally typed language, then we can statically check that a filepath `fp` is indeed a subtype of `string`, $\text{fp} : \text{SafeFilePath} \subseteq \text{fp}' : \text{string}$, but we cannot be statically certain that $\text{fp} : \text{SafeFilePath} \subseteq \text{fp}' : \star$ holds, where $\star$ is the *unknown* type in gradual typing.

5.4 Realising SAFESTRINGS

Having defined SAFESTRINGS and detailed their expression power, we now describe how we implemented them in three languages: TYPESCRIPT, Java, and BOSQUE. We chose TYPESCRIPT because of its powerful and flexible type system and support for both object-oriented and functional programming. Data-as-string is common in both TYPESCRIPT and JavaScript code. JavaScript makes heavy use of *regex* [65]: if the presence of *regex* correlates with string usage, then *JavaScript* also makes heavy use of strings. We needed the type system of TYPESCRIPT for SAFESTRINGS, so we did not use *JavaScript*. We chose Java because of its static type system, popularity, and *OO* programming model. The fact that both retro-fits are similar suggests the potential ease of generalising SAFESTRINGS to other object-based languages, such as Python. For these retro-fitted languages, we designed program transformations Section 5.4.1 that take annotated code and output SAFESTRING code. These program transformations use annotations linked to a core library of definitions. We discuss the construction of these libraries in Section 5.4.2. The core library only captures common use cases, so we also discuss how to make your own SAFESTRINGS in Section 5.4.3, to enrich the core

String foo(@SafeString String str) {...}	⇒	String foo(SafeString str) {...}
@SafeString String foo(String str) {...}	⇒	SafeString foo(String str) {...}
@SafeString String foo(@SafeString String str) {...}	⇒	SafeString foo(SafeString str) {...}
String foo(SafeString str) { return arg1; }	⇒	String foo(SafeString str) { return str.cast(); }
SafeString foo(String str) { return arg1; }	⇒	SafeString foo(String str) { return new SafeString(str); }

Figure 5.5: The program transformation in OO-style pseudo-code. The rewriting of type signatures is relatively easy for arguments. The impact of annotating the return type is more profound, as it involves calling the appropriate constructors.

library with domain specific definitions. BOSQUE does not require a program transformation, as SAFESTRINGS are native; it uses SAFESTRINGS to enforce the structural integrity of messages in RESTful APIs (Section 5.4.4).

5.4.1 Program Transformation

The program transformation, Figure 5.5, for retrofitting languages, is source-to-source. It takes a file and rewrites it into SAFESTRING form. Source level transformation of the code is useful for a number of reasons. For many statically typed languages in common use, *e.g.* Java, the type checker works over the AST rather than compiled byte code. Changing the source is also more transparent, allowing the developer to see, and understand, the emitted code (Listing 5.10). The transformation performs several tasks, as detailed in Figure 5.5. First, it desugars annotations in type signatures to replace them with the appropriate types. If the return type is `string`, but the method body has been performing operations and checks over SAFESTRINGS, then the SAFESTRING needs to be cast back to `string`.

Both the TYPESCRIPT and Java program transformations use a simple textual search and replace. This source-to-source transformation preserves source line numbers. As result, errors share lines in both the original and transformed

```
/*
* We want file to be a SafeString
* @param file : FilePathSafeString
*/
let file : string = '/this/is/a/file.txt';
// the preprocessor performs this rewrite
let file : FilePathSafeString = new FilePathSafeString('/this/is/a
    /file.txt');

/* we need file to be a string again
* @param file : string
*/
api.call.wants.string(file);
// becomes
api.call.wants.string(file.toString());
```

Listing 5.10: Desugaring in TYPESCRIPT. The rewriting is simple and transparent, and is easy to do manually.

files. This facilitates debugging, by eliminating the need to examine or edit the transformed file. Dynamic errors can be made much more informative; the error from the constructor, essentially a parse error, can be incorporated into the error message, as in Listing 5.11.

5.4.2 Constructing a SAFESTRING Library

SAFESTRINGS are not tied to any specific language, and, as such, a SAFESTRING library can be implemented for an existing language. We start by discussing aspects of an implementation based on our experiences doing so for TYPESCRIPT. Firstly, a fixed set of strings forms a trivial yet fundamental SAFESTRING type. Then,

```
/* a typical error message */
t.ts:1:5 : Argument of type 'num' is not assignable to parameter of
    type 'string'

/* improved error message for string parameters */
t.ts:1:5 - error _ = Argument of type 'email' is not assignable to
    parameter
of type 'creditcard'. The input string was supposed to be a credit
  card number, but the input string had the letter 'd' in position 1
      instead
  of a number.
```

Listing 5.11: Dynamically generated error messages now contain information from the parser. *PCRE2* does not provide this type of feedback.

we explore how both Dyck-like grammars and delimited strings are common supertypes, with many SAFESTRINGS inheriting their features. Next, the structure of our Java implementation is explained through a class diagram (Figure 5.6). We discuss using annotations and documentation to allow a developer to specify the SAFESTRING type of a string. Finally, we present a discussion on the importance of an efficient implementation.

To decide what to implement, we first looked at different feature requests on [205]. We also examined the decisions made by [112] for its standard library. Neither of these sources justify their choices. This small survey included css strings, such as font style and colour, zip and post codes, emails, and date strings. When we annotated a Java corpus (Section 5.5.2), we found significant similarities between this initial seed and what we actually used, giving some indication of what might constitute a sufficient core library.

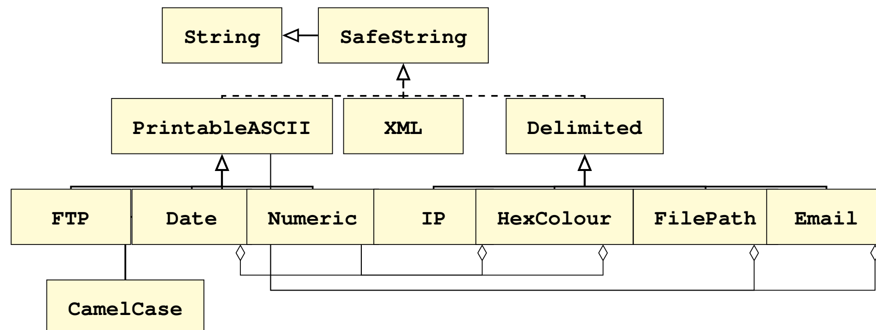

Figure 5.6: Class diagram for all SAFESTRING types used in the Java implementation for SF110 (Section 5.5.2). `String` is the base Java type, which the abstract class `SafeString` inherits from. `PrintableASCII` and `Delimited` all implement `SafeString`. The remaining types inherit from these. The *use of* relationship is also shown, where `PrintableASCII` and `Numeric` are used by other types.

Simple SAFESTRINGS: Word, PrintableASCII and NumericWord. The `Word` SAFESTRING type is any set of fixed strings, *i.e.* its recogniser accepts only inputs directly matching one of its encoded options. This is the same as TYPESCRIPT string literals; it models strings used as enumerations. We did not implement these for TYPESCRIPT as they are already available. `Word` represents a useful generalisation of string literals: instead of matching against an exact pattern, they can match against a regular expression. `PrintableASCII` and `Numeric` SAFESTRINGS are the building blocks of more complex types. `PrintableASCII` contains only those strings with printable characters. This class is useful for those strings where there is little easily discovered structure (*i.e.* natural language strings) but there is a restricted character set. `Numeric` captures those strings which are only numbers. In Figure 5.6, we see that `date`, for example, uses both `PrintableASCII` and `Numeric`

Dyck and Delimited Strings. More complex strings are frequently subtypes of *Dyck strings* [215]. Dyck strings are strings with an underlying Dyck-like grammar, of the form $S \rightarrow$ "("S")S". We relax this to allow other SAFESTRINGS to appear between the balanced delimiters. Another common SAFESTRING supertype is the delimited string. These are strings of fields with a set delimiter, such as CSV. The fields themselves may also be SAFESTRINGS. This is a useful way to encode common structures such as file paths and sort codes for bank accounts. One can represent an email with a separate field for `name`, "@", `domain`, "." and `topleveldomain`, but such a representation is needlessly noisy. A *delimiter* in a SAFESTRING is a substring of the SAFESTRING that always occupies the same position *w.r.t* the other substrings of the string. Both the "@" symbol and the *first* dot "." of an email address are delimiters. They are the 'scaffold' around which the other elements (sub-strings) are disposed. Consider an '*inner-comma*' string, a string of the form $s = s_1 \cdot ',' \cdot s_2$, where $',' \notin (s_1 \cup s_2)$. The simplest suitable representation is a list of length 3. The first element would be the delimiter, ',', while the remainder would be the substrings disposed on either side. This simple schema for delimiters underlies the *DelimitedSafeString* class, which is a parent class for a large number of other SAFESTRINGS. Emails themselves can be modelled in the *DelimitedSafeString* class (Figure 5.7). Delimited strings can have bounds on the number of their fields. A *DelimitedSafeString* with no lower bound (*i.e.* $min = 0$) is a degenerate case. Such a string may be empty, or have one field only.

Java SafeStrings. To overcome the `final` nature of string in Java, we copy the string interface entirely into a new, non-final, class, `SafeString`. The program transformation allows us the pass easily between `String` and `SafeString`, via the `cast()` method and constructor calls. The JVM can optimise `final` classes more

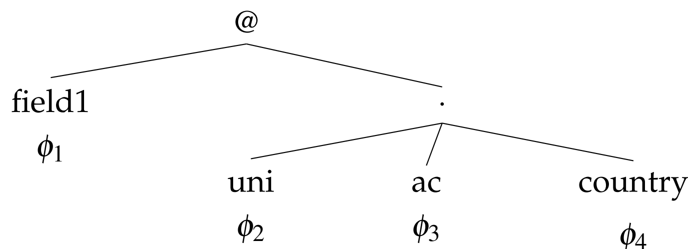

Figure 5.7: Emails are nested delimited strings, with sub-parsers ϕ. The email address "name@uni.ac.pp" can be represented as two different subtypes of delimited string. This increases their subtyping richness, but also makes the internal structure simpler and more efficient: they can be stored as arrays of delimited strings.

efficiently, but we see, in Section 5.5.2, that the cost of adding SAFESTRING is commensurate with the increased safety. Figure 5.6 displays the class diagram of a SAFESTRING implementation in Java. One can see how inheritance plays an important part of the library design, allowing simple yet expressive restrictions of existing types. A number of SAFESTRING types make use of other SAFESTRINGS to restrict the type of fields within their structure. For example, `DateSafeString`, `HexColourString` and `IPSafeString` all use `NumericSafeString`.

Annotations. SAFESTRINGS' Java realisation makes use of the Java annotation language [222]. TYPESCRIPT lacks a similar, convenient, annotation system. It does, however, have a useful documentation language. We use this documentation language to provide the necessary information for the language transformation. We have already seen, in Listing 5.5, how TYPESCRIPT doc strings are used to specialise the base `string` type. BOSQUE does not require annotations, as SAFESTRINGS are part of the language specification.

Efficiency. There is usually a dynamic element to using SAFESTRINGS. The efficiency of this instantiation rests on the parser implementation used. We use parser combinators in the TYPESCRIPT and Java implementations (Section 5.4), as they are an in-language solution and do not require new programs or processes to be added to the development cycle. While it is possible to tailor strings that provoke the worst-case behaviour of a recogniser, as in the regular expression denial of service attack, *ReDOS* [63], we do not consider SAFESTRINGS in this adversarial context in this paper. SAFESTRING operations can be made efficient via incremental parsing, and using parser combinators. Space must also be considered. The space requirements for SAFESTRINGS depends on the size of the AST ^{H}. A well designed encoding can make the SAFESTRING representation more space efficient than that of the `string`. Consider the context-free language $a^n b^n$ of equal numbers of 'a' and 'b' characters. Such a string can be arbitrarily long, but a SAFESTRING need only record the structure and the value of n.

5.4.3 Making Your Own SafeStrings

The procedure for making your own SAFESTRING type is essentially the same for both Java and TYPESCRIPT, so we discuss it in a generalised manner here. BOSQUE has its own means, that nevertheless are similar (Section 5.4.4). To ensure the interoperability of SAFESTRINGS with the `string` type, the base class of SAFESTRING is a copy of the `string` class. That is, all operations, methods and fields available in the base string class are also available in SAFESTRING base class. We have added `tryParse()` and `cast` as abstract methods: these are the only methods that must be implemented when creating a new SAFESTRING subtype. Making your own custom SAFESTRING is as simple as adding a new class to the core library. The program transformation searches within the library for a class of the appropriate name. We use the naming convention "...SafeString" to ensure

```
public class NaturalNumberSafeString extends NumericSafeString {
    public NaturalNumberSafeString(String str) {
        this.value = str;
        validate();
    }
    @Override
    protected void validate() {
        for (char c : this.value.toCharArray()) {
            if (!Character.isDigit(c)) {
              throw new IllegalArgumentException(``NumericSafeString
              contains none numeric character.'');
            }}}}
```

Listing 5.12: A simple SAFESTRING definition. Many definitions need be no more complex than this, as the base SAFESTRING class does a great deal of the difficult work.

that non-SAFESTRING comments are ignored by the program transformation. The annotation must use the same word as the name of the class file in the library. For users who need custom types but do not require advanced techniques such as overridden methods, the bar to SAFESTRING definition is low. Writing the parser, P, and designing ϕ is the single largest programming challenge.

Many SAFESTRINGS are simple, as the core SAFESTRING class does most of the "heavy lifting". We present a simple string subtype, `NaturalNumberSafeString` in Listing 5.12 that shows the minimal effort required to create a basic new type.

Parsing in Java and TYPESCRIPT. We use parser combinators in TYPESCRIPT and PEG parsers in Java. Parser combinators are familiar tools in functional programming, having been first introduced by Burge [38] and extended by others, such

as Hutton [109]. PEG parsers were first introduced by Ford [83] and offer similar advantages to parser combinators. The motivation to use both parser combinators and PEGs was due to the presence of well-maintained libraries in the target languages. Both provide an elegant and declarative method for implementing parsers. They do not require a separate toolchain, thus avoiding integration issues. Parser combinators can allow ambiguity to appear at runtime. This can only be resolved by taking care, in the SAFESTRING definition, to remove ambiguity. Using either PEGs or parser combinators, however, allow SAFESTRINGS to be entirely in-language: a program using SAFESTRINGS is self-contained and does not need anything other than the normal language runtime for deployment. Both PEGs and parser combinators have another extremely useful property that we exploit; they are composable. They give us the separately callable parsers that enable compartmentalised checking (Section 5.3.4). Using a single *PCRE2*, a recogniser for an email address is formidable; given the power to compose and order recognisers however, it is a much simpler proposition:

```
invariantAt  = /@/
name         = /[0-9a-zA-Z]+/
invariantDot = /\./
left         = /[0-9-a-zA-Z-]+/
right        = /[0-9-a-zA-Z-.]+/
```

where the parser sequence is the obvious $(\text{right} \circ \text{invariantDot} \circ \text{left} \circ \text{invariantAt} \circ \text{name})(s)$. Moreover, given a mapping, ϕ, to an appropriate structure:

```
(At '@' (Name string1) (InvariantDot '.' (Left string2) (Right string3)
   ))
```

a change to the sub-string, string_2, requires only dynamically rechecking with the `left` recogniser, as the rest of the structure is unaltered (Section 5.3.4). This is much cheaper than rechecking the entire email address string.

5.4.4 SafeStrings in Bosque

The BOSQUE programming language [129] is a ground up language and tooling co-design project, focused on addressing the challenges developers face when building micro-service or server-less architecture cloud applications. These systems interact heavily via remote calls using RESTful APIs [80] and involve interacting with multiple endpoints implemented using a variety of programming languages. Thus, least-common-denominator data formats are used extensively, including JSON and (latently structured) strings, which can be parsed by any member of the polyglot micro-service confederation and are well suited to sending over diverse forms of communication stacks.

In this development model, the ability to explicitly provide information about the latent structure in string data provides two major benefits. By lifting the implicit structure explicitly into the type system, BOSQUE can drastically improve the quality and usability of any API endpoints it exposes. This makes them easier for other services to consume and helps structure the parsing/validation logic for the service implementer[4]. In addition, by providing explicit structures for the form of data in a string, SAFESTRINGS enable the construction and application of advanced tooling for BOSQUE developers. By using the explicit logic that describes the form of a string, BOSQUE is able to provide a SMT-based generator that constructs valid input strings for symbolic model checking. By using the explicit input string structure, the checker spends drastically less time exploring parsing and validation code and achieves much higher coverage of the actual application logic.

Due to BOSQUE's native SAFESTRING capabilities, it is an easy operation to

[4]For example by catching paths where a string is not properly sanitised/validated and remains of type `String` instead of validated as a `StringOf<Date>`.

extend the available SAFESTRING definitions with new types (Listing 5.13). These simple definitions are a form of light validation and type tagging that prevents confusion in the code and reduces problems with APIs which have multiple string valued parameters, a common cause of argument selection defects [159].

```
typedef L4 = /\w\w\w\w/; \\ SafeString may contain only 4 letters
typedef D4 = /\d\d\d\d/; \\ SafeString may contain only 4 digits

function fss(s1: SafeString<D4>): Bool {
    return s1->string() == ``1234'';
}

fss(SafeString<L4>::as(``abcd''))  // type error L4 incompatible with
    D4
fss(SafeString<D4>::as(``abcd''))  // runtime error 'abcd'
    incompatible with D4
fss(SafeString<D4>::as(``1234''))  // true
```

Listing 5.13: BOSQUE permits the quick creation of simple SAFESTRINGS for input validation and type tagging.

The BOSQUE type system has univariate parametric types and the target domain has a mix of simple string formats that can be easily *regex*-validated, as well as complex data format strings. Thus, the BOSQUE implementation adapts the SAFESTRING concepts slightly to match the needs of the domain and target developers by: (1) providing simpler *regex*-validated `SafeString<T>` where `T` is a regex type definition, and (2) a user defined parser validated `SafeString<T>`, where `T` implements the special `Parsable` trait. As parametric typing is univariate, BOSQUE does not support subtyping on the SAFESTRINGS and uses a simplified equality, in order to be strict on the type and string contents. These simplifications sacrifice some of the power of SAFESTRINGS but retain the key value propositions,

while making them approachable for the target web developer.

5.5 SAFESTRINGS: Evaluation

We evaluate SAFESTRINGS in two ways: case studies and a study of the cost of their adoption. The case studies span all three languages and illustrate the power and expressiveness of SAFESTRINGS. SAFESTRINGS are not free: they require definition and annotation. Section 5.5.2 quantifies these costs. As library implementation details are similar between Java and TYPESCRIPT, we focus on Java in Section 5.5.2.

5.5.1 SAFESTRING Case Studies

We explore the benefits of SAFESTRINGS with examples drawn from real code found on Github or StackOverflow answers. In some cases, we simplified the code for brevity. Historically, the fact that Windows and UNIX choose different file path delimiters has caused many problems. The FilePath study illustrates SAFESTRING inheritance and how SAFESTRINGS can prevent common problems, such as converting file paths across operating systems. Hex colours demonstrate how to redefine a SAFESTRING' operations and specialise them to exploit the structure of the data the SAFESTRING contains. Units of measure often cause issues. Our *css* example shows how a SAFESTRING can make units type checkable and prevent hard to detect mistakes. The final two cases studies show how SAFESTRINGS can eliminate boilerplate validation checks and error handling.

FilePaths. Filepaths are a common feature in many programs and are usually typed as `string`. Filepaths have different formats on different architectures, some using `\\`, some `/`, or even a mixture of the two. Filepaths can also be relative, or absolute. Consider the `join` function, available from `path` TYPESCRIPT node

```
let x = '/foo'; let y = 'more/stuff'; let evil = 'gone\bad';
path.join(x, y, evil);
=> returns '/foo/more/stuff/gone\bad'
```

Listing 5.14: It is easy to make an invalid filepath in most languages, including TYPESCRIPT.

```
let x = '/foo/' : @UnixFilePathSafeString; // valid
let y = 'more/stuff/' @UnixFilePathSafeString;  // valid
let evil = 'gone\bad\' : @WindowsFilePathSafeString; // valid
UnixFilePath.concat(x, y, evil) // bad, but produces a type error.
```

Listing 5.15: We can prevent invalid strings from being passed in as paths statically. We override the `concat` method for FilePath SAFESTRINGS to produce a type safe version of path joining. When SAFESTRINGS are native, as in BOSQUE, then joining is already aware of string subtypes.

module. It is easy to build an invalid path via this method, as in Listing 5.14 [5]. With appropriate type annotations, it is possible to guarantee that this error is caught statically, as in Listing 5.15.

We implement `FilePathSafeString` as a subtype of `DelimitedSafeString`, where the representing structure is a list of `PrintableASCIISafeString`'s (Figure 5.6). The delimiter resides in a separate field. Specialising the delimiter field allows us to generate distinct FilePath subtypes, such as SAFESTRINGS of *Unix* and *Windows* style paths. Moreover, a leading delimiter is optional, creating the possi-

[5] Adapted from a stackoverflow answer at https://stackoverflow.com/questions/41553291/can-you-import-nodes-path-module-using-import-path-from-path/45218692

```
class UnixPathSafeString extends FilePathSafeString {
  separator = '/';
  fields = [``root'', ``then'', ``other'', ``stuff.org''];
  basename = fields.slice(0, -1);
  filename = fields.slice(-1);
}
```

Listing 5.16: Subtyping for filenames via specialising the separator. We can also make the class more usable by setting up aliases for indices into the list containing the `ASCII` typed substrings.

bility of accepting absolute and/or relative paths. By making `UnixPathSafeString` and `WindowsPathSafeString` subtypes of `FilePathSafeString`, we can use them wherever we require a generic FilePath, but control when we require specifically Unix or Windows-style paths. Extracting filename, extension and path are just named projections from the internal representation (Listing 5.16).

Hex Colours. Web-based programming is especially rich in strings. Cascading Style sheets (*css*) is an almost ubiquitous style sheet language that is fundamentally string manipulating. This makes it easy to incorrectly assign data with little that either a type checker or static analyser can do to prevent it. A common usage of *css* is setting colours of different html elements.

We have already seen, in Section 5.1, some of the problems caused by representing colours as strings. Analysing a hex colour string reveals an underlying structure. However, superficial differences in presentation can mean a string comparison of two strings encoding the same hex colour may return inequality. Abstracting a hex colour, however, gives the simple structure `(Colour (Red ℕ) (Green ℕ) (Blue ℕ))`, where each value is 0-255, which ignores presen-

tational differences. Equality is now optionally over the structure of the hex colour, rather than just the raw string. The original strings can, of course, still be compared *as strings*, using SAFESTRINGS' `cast()` operation. The '#' symbol is a SAFESTRING delimiter, so does not need to be stored in the representation: `cast(): string = '#'+ red.toString(16)+ green.toString(16)+ blue .toString(16)`. This `HexColourSafeString` representation, and the fact that it contains only valid hexadecimal colour strings, is much easier to manipulate than `string`. Changing elements in the structure leaves the structure itself, and therefore its type, unaltered. We have already seen an example of this in the `blend()` method of Listing 5.5.

CSS Units. Units in *css*, such as *pixels*, *points* and *picas*, are all strings with an obvious latent structure that vanishes when typed as `string`. This makes it easy to assign the wrong unit to the wrong variable (Listing 5.17). It also makes it difficult for a static analysis to identify these errors. While *css* is not a type checked language, TYPESCRIPT allows inline *css*, exposing it to the TYPESCRIPT type checker. Units are the archetypal `NumericWord` SAFESTRING. A string "1cm" has the simple representation `(CM 1)`. It is trivial to catch the incorrect assignment `let font-size : Point = '1cm'`. Listing 5.17 has a simple mistake, *c* was typed instead of *x* in the unit for `margin`; spotting this error in a dense page of *css* is difficult and time-consuming. A linter may not detect this as an error. A declaration that `margin` takes the union type `(PX | AUTO)` catches the error with minimal programmer effort.

Stripping the unit information and normalising the numerical representation also makes it much easier to define conversion functions that do not expose their complexity to the front-end user (Listing 5.18). This can be achieved through, for example, operator overloading. If operator overloading is not desir-

```
<!-- we assume that margin, width, padding etc have type (PX | AUTO)
    --!>
object { height : 250px; width : 200px; margin: 10pc; margin-bottom:
    10px;
  padding: 5px;}
```

Listing 5.17: Mixing units of measure can lead to topographical errors that are difficult to debug in *css*. SAFESTRING catches this unit error during the normal compilation cycle.

able (TYPESCRIPT, for example, does not encourage it), then TYPESCRIPT's union types can be used to overload a function to choose the correct transformation.

Unneeded Exception Handling. SAFESTRINGS reveal the presence of errors statically, but they can also simplify existing project code. Figure 5.8 shows code for parsing an RGB colour from an input string. This code is not amenable to static checking: exceptions are fundamentally dynamic. Using a `RGBColourSafeString` makes almost all of this uncheckable code superfluous.

The use of SAFESTRINGS turns Figure 5.8 into Listing 5.19. This code can be statically guaranteed to return a value of type `Colour`. The question of how to safely parse an integer within a string has remained open for many years. A StackOverFlow post[6] on this question has been actively discussed for the last 10 years and accumulated 185,000 views at the time of writing. SAFESTRINGS were designed to address most of the cases this thread discusses: strings encoding data with simple structure, like the embedded numbers `NumericSafeString` captures.

Calculator in BOSQUE. Listing 5.20 shows a simple calculator API in BOSQUE using a simple validated `SafeString<CalcOp>` for the structure of the operation

[6]https://stackoverflow.com/questions/1486077/good–way-to-encapsulate-integer-parseint

```
// overloading + for a pixel string
add(operand : Pixel | Picas | Points) {
  if (operand isinstanceof Pixel) { return cm self.value + operand.
     value; }
  else if (operand isinstanceof Picas) { return cm self.value +
     operand.toCm(); }
  // ... etc
}
```

Listing 5.18: The complexity of adding incompatible units is abstracted away from the front-end user.

string and a more complex `StringOf<BigInt>` custom validated format string (via the builtin `BigInt` type). Since the API types ensure that the strings are well formed, the code can safely omit further validation in this case and use the strings directly. This example also shows how literal SAFESTRINGS are constructed and compared in a type safe manner.

5.5.2 The Cost of Adopting SAFESTRINGS

SAFESTRINGS impose a new annotation burden. We break this burden into 3 separate costs; number of annotations to be added to code, number of unique SAFESTRING annotations a developer must be aware of, and the effort to create a new SAFESTRING type. We show these costs are low and do not constitute an obstacle to the adoption of SAFESTRINGS.

We tested SAFESTRINGS on existing Java classes, taken from SF110 [85]. *SF110* is a collection of 110 Java projects selected from source-forge between 2012 and 2014, containing over 23,000 classes. We annotated a subset of *SF110* to assess the annotation burden and the effect of annotations on code quality, as per Section 5.4.

```
public static Color processColour(String colStr, Color defaultColour) {
    Color retC = defaultColour;
    if (colourDef.indexOf(``,'') >= 0) {
      try {
        int index = colourDef.indexOf(``,'');
        int R = Integer.parseInt(colourDef.substring(0,index));
        colourDef = colourDef.substring(index+1,colourDef.length());
        index = colourDef.indexOf(``,'');
        int G = Integer.parseInt(colourDef.substring(0,index));
        colourDef = colourDef.substring(index+1,colourDef.length());
        int B = Integer.parseInt(colourDef);

        retC = new Color(R,G,B);
      } catch (Exception ex) {
        System.err.println(``Problem parsing colour property'');
      }
    }
    return retC;
  }
```

Figure 5.8: Real Java code that parses an RGB string and returns a Colour object. We can remove almost all of this code by the simple inclusion of SAFESTRING annotations.

To do this, we first used Soot [206] to perform a flow dependency analysis on Java classes. This was to identify classes containing at least one method that has a string parameter which is used in a control decision. This is because, in many cases, it is more beneficial to annotate parameters, rather than local variables, in order to increase overall type safety. This analysis identified 1,339 classes.

We chose 500 classes, uniformly at random, from the initial dependency filtering of *SF110*. We manually examined each class to see where strings were being used and to ascertain their structure. Due to the large number of classes to examine, and due to the fact that we anticipate SAFESTRINGS being used in a production setting where the refactoring time budget may be low, we gave ourselves approximately one minute to understand the structure of the string. One minute was envisaged as a stringent time limit: in most cases it was, in fact,

```
public static Color processColour(ColourSafeString colStr, Color
    defaultColour) {
    if (!colStr.isEmpty()) {
      return new Colour(colourDef.r, colorDef.g, colorDef.b);
    } else {
        return defaultColour;
    }
}
```

Listing 5.19: SAFESTRINGS can radically simplify the code that a user needs to write. This is because the language transformation inserts the necessary checks. This checks are correct by construction.

generous. Frequently, determining structure was an easy task, as the parameters usually had informative names (*i.e.* an XML string parameter was called "xml"). On other occasions we had to understand the structure from looking at the control flow of the method. Some strings could not be assigned a type. This was due to a lack of interaction within the function body. For example, an existence, or non-null check of a string parameter called `key`, while suggestive, does not provide enough information to decide its grammar.

We observed 28 unique SAFESTRING types, among the latently structured strings in the 500 classes, excluding strings that we were unable to adequately charaterise within our time limit. A developer with domain specific knowledge could easily add the required SAFESTRINGS, building on the core library definitions. Next, we chose, without replacement, a further thirty classes from the initial dependency filtering. These thirty classes only used 10 SAFESTRING types, as detailed in Figure 5.9. The overlap between the assumptions made by Rosie [112] and types found in our subset of *SF110* is substantial, but not total. For example,

```
typedef CalcOp = /negate|add|sub/;

entrypoint function main(op: SafeString<CalcOp>,
    arg1: StringOf<BigInt>, arg2?: StringOf<BigInt>): StringOf<BigInt>
    requires release (op == CalcOp'add' || op == CalcOp'sub') ==> arg2
        != none;
{
    if(op == CalcOp'negate') {
        return SafeString<BigInt>::stringify(-1 * BigInt::parse(arg1))
            ;
    }
    else {
        ...;
    }
}
```

Listing 5.20: A BOSQUE calculator API endpoint using SafeString and StringOf validated string types.

we did not anticipate a `CamelCase` subtype. Given that `CamelCase` is the recommended style for variable naming in Java, it seems likely that this string subtype is language dependent. Other programming languages might also have language dependent types.

Usage of SAFESTRINGS, within the thirty classes, follows a Pareto distribution, with α being close to 1. This suggests that, to benefit from SAFESTRINGS, a developer need only use a handful of types. The three most common SAFESTRING types account for over 50% of required annotations. Within the 10 types in Figure 5.9, there was an average of 35 source lines of code (SLOC), excluding imports. This indicates relatively little effort is required of a developer to create a

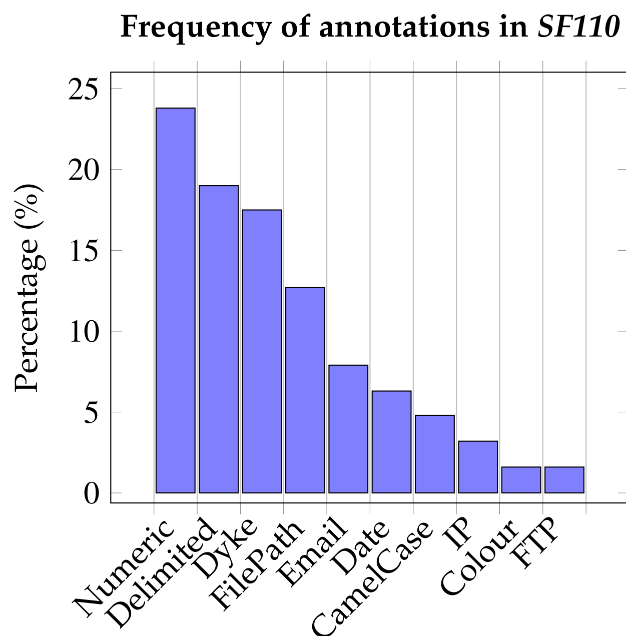

Figure 5.9: Type frequency in the 30 classes sampled from SF110.

new SafeString type.

The left-hand plot in Figure 5.10 shows the average worst case construction times for SafeStrings. Measuring constructor performance in Java is difficult. We ran each constructor a million times, with string inputs randomly taken from lists of prepared (correct) string inputs, up to a length of 128 characters. To ensure that the loop was fully executed, we disabled the JVM's JIT compiler. The JVM JIT compiler is a sophisticated piece of software, and is quick to remove 'dead code', such as unused constructor calls. As a result, the numbers represent the very first time the JVM encounters a SafeString type. As the parsers are static methods, once the parser is JIT'ed, the cost of further instantiations is less. The outlier is `EmailSafeString`. This is likely due to the relative complexity of the

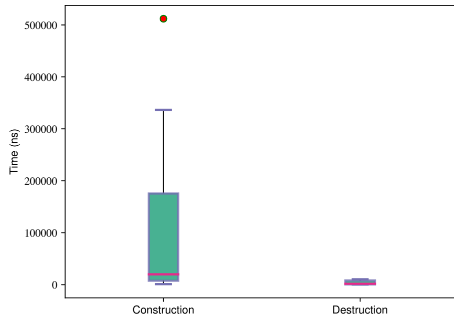

Figure 5.10: The average *worst case* ϕ construction time for the SAFESTRINGS in our Java library.

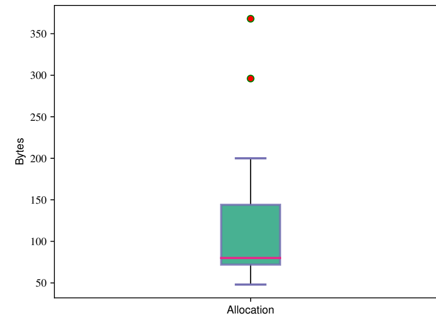

Figure 5.11: Average *worst case* memory consumption for SAFESTRINGS.

grammar, but might also indict that the implementation needs optimisation. The right-hand plot in Figure 5.10 shows the average worst case lstinlinetoString times for the Java library. Once the `toString()` method has been called once, the result is memoised in the SAFESTRING object, hence the construction time need be paid only once. For some simple strings, such as `BinarySafeString`, the input value is trimmed of white space and stored directly, assuming it passes the membership test.

Figure 5.11 shows average memory impact per SAFESTRING. We used **JOL**, the Java Object Library, to obtain information about the size of our SAFESTRING objects. The parser implementations account for the majority of increased memory usage. Again, as parsers are shared per SAFESTRING, this increased memory usage is paid only once per SAFESTRING type. The outliers are those with complex `parboiled` parsers, (`CamelCaseSafeString` and `EmailSafeString`). This could doubtless be make more efficient, but we have opted for readability and correctness as a default.

5.6 Related Work

The problems associated with strings are well known and long standing. There have been many suggestions for string safety from both the research community and industry. We focus on each in turn. We start with common design patterns and partial solutions. These are industry practices, so may lack theoretical rigour or be informally applied. We then move onto attempts to model strings in type systems, before considering syntax embedding and DSLs. This work focuses on string sanitisation, rather than type checking for strings. SAFESTRINGS require type annotations, we explore how type specific languages might be a means to introduce elegant syntax. Finally, we consider techniques that might make library construction easier, such as using SMT solvers to determine operation closure properties and subtype relation construction.

Design Patterns. There are a number of informal approaches to increased string type safety which make for reasonably usable design patterns. All of these require more effort from the user than the simple annotation language of SAFESTRINGS. *Type aliasing* is a simple means for *ad hoc* string typing [197]. This is supported in many languages, but its primary use case is often for documentation purposes rather than type safety. In TYPESCRIPT, for example, one writes `type Name = string`. From the perspective of enforcing invariants however, the utility of aliasing is limited because it does not enforce type (in-)equality. It does not prevent static type errors of the kind `gmail : Gmail = "not a gmail address"`, where `Gmail` is some previously declared type alias for `string`. If a language has *newtypes*, greater type safety can be achieved. These wrap an existing type. A *newtype* is different from an alias in that it is exposed to the type checker and can be used to enforce inequality. Object-oriented languages emulate these by

```
class Monolithic {
  // recogniser check on input
  constructor(s) {
    if p s then mono = s else error ``bad parse'';
  }
  Bool p (String s) : /regex/ check
  String mono;
}
```

Listing 5.21: An object wrapped string.

wrapping a primitive type in an object or using an interface [197]. This object wrapper pattern can be validated with a recogniser. A simple instance of this pattern is explored in [84] (Listing 5.21). The constructor is augmented with a *regex* check for string membership. The constructor fails with an error if the string is not in the language expressed by the *regex*.

Since version 1.8, TYPESCRIPT has had *string* and *number* literal types [197]. The original pull request for string literals [161] summarises them as: "A string literal type is a type whose expected value is a string with textual contents equal to that of the string literal type." A string literal type can only be assigned the exact value specified in that type. String literal types can be used with other features of the type system, most notably *union types*, to create (finite) enumerable sets of strings. This allows them to act as type guards in pattern-matching in a similar way to type constructors in pattern-matching within the ML family. *Regex*-extended string literals relax the need for exact matching, making string literal types even more expressive. SAFESTRINGS subsume these approaches, adding extra functionality as well as acting as a relaxation for exact matching in pattern guards. TYPESCRIPT 4.0 has increased the flexibility and type safety of strings by

featuring literal narrowing, allowing constant strings to have their own unique type. They do not have the expressive range of SAFESTRINGS, which can easily implement this feature, but can also allow for subtyping and interoperability with strings. Moreover, these features in TYPESCRIPT are changes made at the level of the compiler, whereas SAFESTRINGS and be retrofitted to a language without change to the implementation.

SAFESTRINGS are an example of the "parse, don't validate"[7] philosophy commonly found in languages such as Haskell. Validation, instead of parsing, is one of the weaknesses of the regex-validated strings approach. Haskell has *typeclasses*. In particular, `Read` and `Show` provide (and check for) (de)-serialisabilty of an algebraic data type. They are the logical way to implement SAFESTRINGS. The core of SAFESTRINGS's contribution is the contract that it imposes on any realisation. Notably, Equation (5.1) ensures sound conversion between the ADT and string representations of the data. SAFESTRINGS also do not alter the string interface, meaning they are drop-in replacements for string. Doing this latter operation in Haskell is not possible without the use of a program transformation (Section 5.4.1), as Haskell does not support subtyping.

One can use parametric polymorphism to mimic some of the advantages of a program transformation, but at the risk that a function's type signature becomes too general, *i.e.* the desired `foo :: Email -> String` needs to become `foo :: (Read a)=> a -> String`, which accepts far more than the intended signature. To support easier type checking for SAFESTRINGS in Haskell, where tools give real time type information that might be confounded by a pre-compilation program transformation, future work will look at embedding SAFESTRING type annotations within a refinement type system, such as Liquid Haskell. Finally, most OO

[7]https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/ accessed Dec 2020

languages do not have equivalents of typeclasses: our annotation-based method overcomes this problem.

Strings in Type Systems. Recognising the problem of structured input encoded as strings, [107] introduce regular expression types for XML. These encode the structure of the XML at the type level, combining typing judgements via operators inspired by those of regular languages. This has the advantage of providing expressive types for each XML string, creating a family of XML-types for all schema. SAFESTRINGS do not aim for this degree of expressiveness, choosing instead to model valid XML as one type. Of course, one could subtype to create a similar family of types, but this would not be as syntactically clean. Subtyping with regular expression types is not modelled in the host language, but reduces to the inclusion problem for tree automata. In contrast, SAFESTRING is eminently practical and designed to be entirely within language. Their regular expression types only apply to XML and would have to be implemented for a target language. Finally, and most importantly, regular expression types are no longer strings, and no longer easily inter-operate with strings; the programming model has changed.

PCRE2-validated string types of [205] allow for a high degree of type safety *w.r.t* to strings. A type declaration takes the form: `type CssColor = /^#([0-9a-f]{3}|[0-9a-f]{6})`, with assignment being `let fontColor : CssColor = #000`. SAFESTRINGS are strictly more expressive, however, in that they are not tied to one form of membership test (*i.e. PCRE2*): arbitrary logic can be included in the recogniser for a SAFESTRING. SAFESTRINGS also offer elegant support for subtyping and operations over strings, which *PCRE2*-validated strings do not. There are expressive alternatives to using *regexes*.

The *Rosie Pattern Language* (RPL) of [112] is a replacement for more familiar *regexes*. RPL is based on *Parsing Expression Grammars*, or PEG [83]. It should be

possible to use RPL as an alternative for parser combinators. RPL, by itself, is not an alternative for SAFESTRINGS, as it is simply a pattern matching language, rather than a means to expose structure to a type checker.

CLOTHO [70] is a taint analysis tool for strings. It uses a hybrid approach, with a static analysis designed to detect statements likely to represent inappropriate string handling. It combines this with a dynamic analysis which generates patches for the incorrect code in the event of execution. CLOTHO is not a type based approach, though some of its techniques would be adaptable to enable SAFESTRING inference. Automatic patch generation should see improved results in the presence of more precise type information.

Syntax Embedding and DSLs. Work on syntax embedding, such as [33], aims to make code immune to injection attacks, by construction. They embed the grammar of a guest language, such as SQL, into the host language. Immediately, they have moved away from a string programming model. They generate code that maps the guest language to the host language, escaping functions as appropriate. This approach has different intentions from SAFESTRINGS, where we are concerned more with the shallow embedding of *data* for the type checker, rather than the deep embedding of executable code for safety. SAFESTRINGS can be used to do a shallow pass over SQL, preventing syntactically invalid strings from being executing. Syntax embedding is akin to compilation, a heavyweight solution compared to SAFESTRINGS. Again, and most importantly, any conception of the input string as `string` is lost. The programming model has changed, whereas SAFESTRINGS preserves the experience of string-based programming.

Syntax. SAFESTRINGS need type annotations for TYPESCRIPT and Java. It is difficult to retroactively add syntactic features to these languages. We solve this using a preprocessing stage. BOSQUE already has native syntax

for SAFESTRINGS. [147, 148] introduce *type specific languages* and *reasonably programmable literal notation*, which is a principled approach to adding new literal notation to existing programming languages. As such, their approach could be used to provide direct syntax support for retrofitting SAFESTRINGS, without the need for a separate program transformation pass. For the moment, their work is limited to Reason, an alternative front-end for OCaml. As we require explicit annotations, OCaml and Reason are not a suitable testing bed for SAFESTRINGS. Future work could hopefully make use of this powerful approach to obviate the need for an annotation language and independent preprocessor.

Operations. Writing a SAFESTRING library requires effort. In addition to writing the parsers, decisions need to be made about methods. SAFESTRINGS provide a mechanism for lifting string operations, but whether the lifted operation is closed under the SAFESTRING is difficult to determine and may require computer assistance. The question of the decidability of string theory, *i.e.* the problem of automatically solving string constraints, has seen a recent renewal of interest in the research community [2, 3, 87, 45]. This is doubtless due to the recognition of the ubiquity of string manipulating programs in many programming domains. A large amount of recent work has focused on the development of practical string solvers. The list of SMT solvers handling at least a part of string theory includes Z3-str [226], CVC4 [124] and Stranger [221]. The primary use case for these solvers is in symbolic analysis, as discussed in [115] and [39]. The need to recheck after an operation over a SAFESTRING is one of its greatest costs, even allowing for the fact that, in general, we only require rechecking of sub-trees.

5.7 Conclusion

We have presented SAFESTRINGS, a language-agnostic approach to type safety for strings. SAFESTRINGS require no special language mechanisms, requiring only an ability to encode structures and recognisers. SAFESTRINGS translate a structure over which the type checker has no particular knowledge (*i.e.* `string`) into a form that exposes rich latent structure. SAFESTRINGS are applicable either statically or dynamically. SAFESTRINGS squeeze more invariants out of simple type systems by treating strings as algebraic structures. The complexity of advanced language features, such as dependent types and GADTs, are not required. A simple type system can go further by putting the burden of representation on the producer of library code, rather than the consumer. We have also presented instantiations of SAFESTRINGS in Java, BOSQUE, and TYPESCRIPT, showing the relative ease with which they can be encoded and used.

Chapter 6

Conclusion

This chapter summarises the overall conclusions of this thesis and how the presented work addresses the problems under investigation. It also discusses how the approaches presented can be extended, improved and enhanced in future work.

6.1 Contributions and Summary

The main contributions of the thesis are:

- **Empirical study on ranked information flow**
  We conduct the first empirical study into ranked information flow. Ranking information flow relies on relative magnitude of information movement, rather than precise information estimates. We have presented empirical evidence that ranking of information flow is more efficiently computable than exact estimation. Over nearly 800 different functions, we show that the ranking is 95% stable after 3 hours of fuzzing. This is equivalent to just 40% of the minimum number of samples required by similar tools.

- **A new Information Theoretic Measure**
  We introduce *FlowForward* (Definition 3.3.1), a new information theoretic

measure suitable for ranking. It is bounded in the interval $[0, 1]$. Due to its normalisation properties, it stabilises rapidly, even when the mutual information estimate is still changing (Figure 4.13).

- **Software Engineering Uses for RIF**

  We introduce *information contour maps* (Section 3.2) to visualise information movement within the SUT. The ICM is a colour-coded callgraph, in which the colours represent relative flow size. We present various uses and interpretations for ICMs, including, but not limited to, security, refactoring and test case adequacy analysis.

- **RIFFLER**

  We present a prototype RIF and ICM tool generation tool, RIFFLER, which decorates a PYTHON program an captures input/output pairs at the function level. RIFFLER calculates *FlowForward* at a set time increment. It can colour code a callgraph provided in GRAPHVIZ *dot* format, or provide detailed analysis in the terminal. RIFFLER adds and removes PYTHON function decorators semi-automatically and integrates seamlessly into a standard fuzzing campaign.

- **OAST**

  We present the first optional type system for security than does not use a dynamic cast calculus. OAST replaces the dynamic monitoring of traditional gradual typing with a testing phase that utilises RIF. OAST solves the brittleness problem of security typing by providing the software engineer with detailed risk information embedded inside provably secure components. We introduce and prove the *confinement property* Definition 4.6.1 for OAST, that states that any errors that occur must originate in untyped regions

of the code. We present a prototype PYTHON implementation of OAST, highlighting the flexibility of the system, but also showing shortcomings in the prototype; specifically its failure to satisfy confinement.

- **SAFESTRINGS**
  We present SAFESTRINGS, a novel type embedding for increased type safety for strings. SAFESTRINGS use the native capabilities of an object-based type system, and a program transformation, to enable strings to be labelled with more precise types. These types serve to reduce the available entropy for a string: the developer can specify that a string s : string is s : email $\subseteq$ string. We present a SAFESTRING library in Java, and detail various case studies highlighting their use. We report performance characteristics and the annotation burden for a subset of the Java SF110 corpus of programs. We also analysis the type of strings used in SF110, and discover that they follow a broadly *Zipfian* distribution. SAFESTRINGS have already been incorporated natively into a language from MICROSOFT, BOSQUE. SAFESTRINGS solve the subtyping problem of *regex*-validated strings. By using annotations and a program transformation, one can retrofit a language with SAFESTRINGS.

6.2 Future Work

Much work remains to be done to further the approaches detailed in this thesis. The main thrust of this thesis has been on novelty: to scale the approaches to effective software engineering tools required a larger engineering effort. The techniques described in this thesis are intended to be practical: we have deliberately abandoned formal proofs of properties in favour of probabilistic guarantees whenever a formal system presents itself as too inflexible. This is most obvious in OAST, but it also holds for SAFESTRINGS, where a SAFESTRING substring can be

cast, at any point, back up to `string`, thus voluntarily discarding information.

RIFFLER works on pure PYTHON programs only. However, there is nothing inherent in the approach that limits it in this manner. Future versions of RIFFLER will seek to address the problem of adding annotations automatically to C. This will allow RIFFLER to be lifted to handle large scale software such as *numpy* and *pandas*. The problem of false constant functions (Section 3.4) is likely to increase for these programs. A custom fork of the *dill* library, with a controllable recursion depth, would go a long way to solving this problem. RIFFLER functions in two parts: a PYTHON specific part, and a RIF calculator. Future work will decouple these two sections, allowing for new language specific front-ends to be added.

We have tested RIFFLER via fuzzing, but an analysis of the callgraphs generated suggests that fuzzing may not be the most efficient way to compute RIF. While fuzzing has the virtue of requiring little setup, and little input from the user, it also wastes resources on some functions, calling them far more than required, and fails to adequately sample other functions. For the point of view of continuous integration, a 3 hour fuzzing budget may be too large. Future work will examine the benefits of different testing methods, such as adapting traditional test suites. It is not possible to use most test suites directly, due to their inability to generate many random inputs. Ideally, it would be possible to rewrite test suites quasi-automatically to include input generators, and not just user-provided hard-coded values.

Linking specific language features to particular types of *FlowForward* would greatly simplify the interpretation of ICMs. While every attempt has been made to make an ICM easy to understand, requiring no specific knowledge of information theory, exact interpretation is more difficult. Future work will examine specific constructs to measure their average *FlowForward*. With this knowledge, it will be

possible to do outlier analysis. The hope is that such anomalies as Figure 3.1 will be identified automatically without user intervention.

User guidance needs to be developed to assist a user in interpreting the various metrics that RIFFLER generates. This is a non-trivial task. An ICM requires interpretation, easiest when it either provokes cognitive dissonance, or cognitive agreement. For example, we would expect a good hash function to have high *FlowForward*, and an ICM could indicate the quality of a hash function. But this is only true if the user has an expectation of the behaviour of a hash function. For non-obvious code, the interpretation may be non obvious. There are various methods we could explore to make interpretation easier, such as looking at natural language channels in code (variable names for example may be semantically loaded) or relying on lightweight annotations from a developer.

Although we focus on the stability of ranking in Chapter 3, precise ranking are not essential for all uses. A broader treatment ranking as just *low, middle* and *high* could provide a great deal of useful information. Indeed, as can be seen in Figure 4.13, the interesting behaviour is not even whether the *FlowForward* is high or low, but whether it is flat or changing when plotted. Functions which exhibit this “drifting” *FlowForward* behaviour are doing something interesting with their information. Further work is required to understand what this thing is, and how to exploit this knowledge for *e.g.* test case prioritorisation. As an example, changing input distribution when *FlowForward* is flat should increase the probability of provoking new behaviours in code. This could potentially increase testing efficiency, or even be a complimentary metric to line coverage. Stability of ranking, where is it important, is expressible in different ways. For example, it may be more intuitive for users to characterise uncertainty in ranking in terms of probabilities: *e.g.* 0.01 probability of any ranking being more than 3 out. User

studies or surveys may go some way to answering questions about information presentation in the absence of information theory knowledge.

While non-functional properties have not been the focus of this work, there is no reason *in principle* why these cannot be considered as outputs for RIF analysis. Indeed, there is the potential for rich interplay between *FlowForward* rankings of functional and non-functional properties: such an interplay may assist in automating the search for dissonance that so assists in ICM interpretation.

We have presented a formal description of OAST for a simple λ-calculus. This only hints at the full utility of the approach. Unfortunately, no existing PYTHON type checker allows for the creation of custom types to be embedded into the checker. This limits to power of the PYTHON prototype, which is far too brittle to be used in production. The type checker MYPY supports plugin type checkers. It might be possible to include a security type checker via a plugin, though there is no evidence that custom types are introducible via this method. A more realistic approach would be a security orientated fork of MYPY.

SAFESTRINGS have a mature library for Java, and a prototype library for TYPESCRIPT. At time of submission, work is already far along in demonstrating the power of SAFESTRINGS for improved test case generation and improved coverage for Java programs and RESTful APIs. This work does not form part of this thesis. Future work will focus on improving the efficiency of SAFESTRINGS, the bottleneck being the quality of the parsers. We will also introduce a simple schema for creating SAFESTRINGS within a Java program, via types already existing in the SAFESTRING library. In this way, a *regex* could be conveniently embedded in the SAFESTRING framework, gaining the subtyping power lacking in *regexes*, but with the easy extensibility of writing a simple *in program* declaration.

Bibliography

[1] Abadi, M., Cardelli, L., Pierce, B., Plotkin, G.: Dynamic typing in a statically-typed language. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 213–227. POPL '89, ACM, New York, NY, USA (1989), http://doi.acm.org/10.1145/75277.75296

[2] Abdulla, P.A., Atig, M.F., Chen, Y., Diep, B.P., Holík, L., Rezine, A., Rümmer, P.: Flatten and conquer: a framework for efficient analysis of string constraints. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017. pp. 602–617. ACM, New York, NY, USA (2017). https://doi.org/10.1145/3062341.3062384, https://doi.org/10.1145/3062341.3062384

[3] Abdulla, P.A., Atig, M.F., Chen, Y., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: String constraints for verification. In: Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. pp. 150–166. New York, NY, USA (2014), https://doi.org/10.1007/978-3-319-08867-9_10

[4] Ahmed, A., Findler, R.B., Siek, J.G., Wadler, P.: Blame for all. SIGPLAN Not. **46**(1), 201–214 (Jan 2011), http://doi.acm.org/10.1145/1925844.1926409

[5] Ahmed, A., Jamner, D., Siek, J.G., Wadler, P.: Theorems for free for free: Parametricity, with and without types. Proc. ACM Program. Lang. **1**(ICFP), 1–28 (Aug 2017), http://doi.acm.org/10.1145/3110283

[6] Alvim, M.S., Chatzikokolakis, K., Palamidessi, C., Smith, G.: Measuring information leakage using generalized gain functions. In: 25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012. pp. 265–279 (2012), https://doi.org/10.1109/CSF.2012.26

[7] Alvim, M., Chatzikokolakis, K., McIver, A., Morgan, C., Palamidessi, C., Smith, G.: The Science of Quantitative Information Flow. Information Security and Cryptography, Springer International Publishing (2020), https://books.google.co.uk/books?id=jJH-DwAAQBAJ

[8] de Amorim, A.A., Fredrikson, M., Jia, L.: Reconciling noninterference and gradual typing. In: Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science. p. 116–129. LICS '20, Association for Computing Machinery, New York, NY, USA (2020), https://doi.org/10.1145/3373718.3394778

[9] Amtoft, T., Banerjee, A.: Information Flow Analysis in Logical Form, pp. 100–115. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)

[10] Androutsopoulos, K., Clark, D., Dan, H., Hierons, R.M., Harman, M.: An analysis of the relationship between conditional entropy and failed error propagation in software testing. In: Proceedings of the 36th International

Conference on Software Engineering. pp. 573–583. ICSE 2014, Association for Computing Machinery, New York, NY, USA (2014)

[11] Anubhav722: ssh decorator (2018)

[12] Askarov, A., Hunt, S., Sabelfeld, A., Sands, D.: Termination-insensitive noninterference leaks more than just a bit. In: Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security. pp. 333–348. ESORICS '08, Springer-Verlag, Berlin, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-88313-5_22

[13] Askarov, A., Sabelfeld, A.: Security-typed languages for implementation of cryptographic protocols: A case study. In: Computer Security - ESORICS 2005, 10th European Symposium on Research in Computer Security, Milan, Italy, September 12-14, 2005, Proceedings. pp. 197–221 (2005), https://doi.org/10.1007/11555827{_}12

[14] Askarov, A., Sabelfeld, A.: Tight enforcement of information-release policies for dynamic languages. In: Proceedings of the 2009 22Nd IEEE Computer Security Foundations Symposium. pp. 43–59. CSF '09, IEEE Computer Society, Washington, DC, USA (2009), http://dx.doi.org/10.1109/CSF.2009.22

[15] Austin, T.H., Flanagan, C.: Efficient purely-dynamic information flow analysis. SIGPLAN Not. **44**(8), 20–31 (Dec 2009), http://doi.acm.org/10.1145/1667209.1667223

[16] Austin, T.H., Flanagan, C.: Efficient purely-dynamic information flow analysis. In: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security. pp. 113–124. PLAS

'09, Association for Computing Machinery, New York, NY, USA (2009), https://doi.org/10.1145/1554339.1554353

[17] Austin, T.H., Flanagan, C.: Permissive dynamic information flow analysis. In: Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security. pp. 1–12. PLAS '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1814217.1814220

[18] Austin, T.H., Flanagan, C.: Permissive dynamic information flow analysis. In: Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security. PLAS '10, Association for Computing Machinery, New York, NY, USA (2010), https://doi.org/10.1145/1814217.1814220

[19] Austin, T.H., Flanagan, C.: Multiple facets for dynamic information flow. SIGPLAN Not. **47**(1), 165–178 (Jan 2012), http://doi.acm.org/10.1145/2103621.2103677

[20] Backes, M., Köpf, B., Rybalchenko, A.: Automatic discovery and quantification of information leaks. In: 2009 30th IEEE Symposium on Security and Privacy. pp. 141–153 (2009)

[21] Basharin, G.: On a statistical estimate for the entropy of a sequence of independent random variables. Theory of Probability and Its Applications **4**, 333–336 (1959)

[22] Basharin, G.P.: On a statistical estimate for the entropy of a sequence of independent random variables. Theory of Probability and its Applications **4**, 333–336 (1959), translated from Russian

[23] Bauman, S., Bolz-Tereick, C.F., Siek, J., Tobin-Hochstadt, S.: Sound gradual typing: Only mostly dead. Proc. ACM Program. Lang. **1**(OOPSLA), 1–24 (Oct 2017), http://doi.acm.org/10.1145/3133878

[24] Bell, D.E., Lapadula, L.J.: Secure computer system: Unified exposition and multics interpretation (1976)

[25] Bell, D.E., Padula, L.J.L.: Secure computer system: Unified exposition and multics interpretation (1976)

[26] Biondi, F., Enescu, M.A., Heuser, A., Legay, A., Meel, K.S., Quilbeuf, J.: Scalable approximation of quantitative information flow in programs. In: International Conference on Verification, Model Checking, and Abstract Interpretation. pp. 71–93. Springer (2018)

[27] Biondi, F., Legay, A., Traonouez, L.M., Wasowski, A.: QUAIL: A Quantitative Security Analyzer for Imperative Code, pp. 702–707 (2013)

[28] Böhme, M., Manès, V.J.M., Cha, S.K.: Boosting fuzzer efficiency: An information theoretic perspective. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2020, Association for Computing Machinery, New York, NY, USA (2020), https://doi.org/10.1145/3368089.3409748

[29] Bonnaire-Sergeant, A., Davies, R., Tobin-Hochstadt, S.: Practical Optional Types for Clojure, pp. 68–94. Springer Berlin Heidelberg, Berlin, Heidelberg (2016), https://doi.org/10.1007/978-3-662-49498-1{_}4

[30] Boudol, G.: Secure Information Flow as a Safety Property, pp. 20–34. Springer Berlin Heidelberg, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-01465-9{_}2

[31] Bracha, G.: Pluggable type systems. In: In OOPSLA'04 Workshop on Revival of Dynamic Languages (2004)

[32] Bracha, G., Griswold, D.: Strongtalk: Typechecking smalltalk in a production environment. SIGPLAN Not. **28**(10), 215–230 (Oct 1993), http://doi.acm.org/10.1145/167962.165893

[33] Bravenboer, M., Dolstra, E., Visser, E.: Preventing injection attacks with syntax embeddings. Science of Computer Programming **75**(7), 473–495 (2010)

[34] Broberg, N., van Delft, B., Sands, D.: Paragon for Practical Programming with Information-Flow Control, pp. 217–232. Springer International Publishing, Cham (2013), http://dx.doi.org/10.1007/978-3-319-03542-0{_}16

[35] Broberg, N., Sands, D.: Flow Locks: Towards a Core Calculus for Dynamic Flow Policies, pp. 180–196. Springer Berlin Heidelberg, Berlin, Heidelberg (2006), http://dx.doi.org/10.1007/11693024{_}13

[36] Broberg, N., Sands, D.: Paralocks: Role-based information flow control and beyond. SIGPLAN Not. **45**(1), 431–444 (Jan 2010), http://doi.acm.org/10.1145/1707801.1706349

[37] Buiras, P., Stefan, D., Russo, A.: On dynamic flow-sensitive floating-label systems. CoRR **1507.06189** (2015), http://arxiv.org/abs/1507.06189

[38] Burge, W.H.: Recursive programming techniques. Addison-Wesley Pub, Reading, Massachusetts (1975)

[39] Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: Exe: Automatically generating inputs of death. In: Proceedings of the 13th ACM Conference on Computer and Communications Security. pp. 322–335. CCS '06, ACM, New York, NY, USA (2006), http://doi.acm.org/10.1145/1180405.1180445

[40] Catalin Cimpanu: Backdoored python library caught stealing SSH credentials (2018), https://www.bleepingcomputer.com/news/security/backdoored-python-library-caught-stealing-ssh-credentials/, [Online; accessed 16-Aug-2021]

[41] Chandra, D., Franz, M.: Fine-grained information flow analysis and enforcement in a java virtual machine. In: ACSAC (2007)

[42] Chapman, R., Hilton, A.: Enforcing security and safety models with an information flow analysis tool. Ada Lett. **XXIV**(4), 39–46 (Nov 2004), http://doi.acm.org/10.1145/1046191.1032305

[43] Chatterjee, K., Henzinger, T.A.: The complexity of quantitative information flow problems. In: In CSF. pp. 205–218 (2011)

[44] Chatzikokolakis, K., Smith, G.: Refinement metrics for quantitative information flow. In: The Art of Modelling Computational Systems: A Journey from Logic and Concurrency to Security and Privacy, pp. 397–416. Springer (2019)

[45] Chen, T., Chen, Y., Hague, M., Lin, A.W., Wu, Z.: What is decidable about string constraints with the replaceall function. Proceedings of the ACM on Programming Languages **2**(POPL), 3 (2017)

[46] Cherubin, G., Chatzikokolakis, K., Palamidessi, C.: F-BLEAU: Fast Black-Box Leakage Estimation. In: S&P 2019 - 40th IEEE Symposium on Security and Privacy. pp. 835–852. IEEE, San Francisco, United States (May 2019), https://hal.archives-ouvertes.fr/hal-02422945

[47] Chong, S., Liu, J., Myers, A.C., Qi, X., Vikram, K., Zheng, L., Zheng, X.: Secure web applications via automatic partitioning. SIGOPS Oper. Syst. Rev. **41**(6), 31–44 (Oct 2007), http://doi.acm.org/10.1145/1323293.1294265

[48] Chong, S., Vikram, K., Myers, A.C.: SIF: Enforcing confidentiality and integrity in web applications. In: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium. pp. 1–16. SS'07, USENIX Association, Berkeley, CA, USA (2007), http://dl.acm.org/citation.cfm?id=1362903.1362904

[49] Chothia, T., Kawamoto, Y., Novakovic, C.: Leakwatch: Estimating information leakage from java programs. In: Kutyłowski, M., Vaidya, J. (eds.) Computer Security - ESORICS 2014. pp. 219–236. Springer International Publishing, Cham (2014)

[50] Chung, B., Li, P., Nardelli, F.Z., Vitek, J.: KafKa: Gradual Typing for Objects. In: Millstein, T. (ed.) 32nd European Conference on Object-Oriented Programming (ECOOP 2018). Leibniz International Proceedings in Informatics (LIPIcs), vol. 109, pp. 1–24. Schloss Dagstuhl–Leibniz-Zentrum fuer Infor-

matik, Dagstuhl, Germany (2018), http://drops.dagstuhl.de/opus/volltexte/2018/9217

[51] Cimini, M., Siek, J.G.: The gradualizer: A methodology and algorithm for generating gradual type systems. SIGPLAN Not. **51**(1), 443–455 (Jan 2016), http://doi.acm.org/10.1145/2914770.2837632

[52] Cimini, M., Siek, J.G.: Automatically generating the dynamic semantics of gradually typed languages. SIGPLAN Not. **52**(1), 789–803 (Jan 2017), http://doi.acm.org/10.1145/3093333.3009863

[53] Clark, D., Hierons, R.M.: Squeeziness: An information theoretic measure for avoiding fault masking. Information Processing Letters **112**(8-9), 335–340 (2012)

[54] Clark, D., Hierons, R.M., Patel, K.: Normalised squeeziness and failed error propagation. Information Processing Letters **149**, 6–9 (2019)

[55] Clark, D., Hunt, S., Malacaria, P.: Quantitative analysis of the leakage of confidential data. Electronic Notes in Theoretical Computer Science **59**(3), 238–251 (2002)

[56] Clark, D., Hunt, S., Malacaria, P.: Quantitative information flow, relations and polymorphic types. Journal of Logic and Computation **15**(2), 181–199 (2005)

[57] Clark, D., Hunt, S., Malacaria, P.: A static analysis for quantifying information flow in a simple imperative language. Journal of Computer Security **15**(3), 321–371 (2007)

[58] Clarkson, M.R., Chong, S., Myers, A.C.: Civitas: Toward a secure voting system. In: Proceedings of the 2008 IEEE Symposium on Security and Privacy. pp. 354–368. SP '08, IEEE Computer Society, Washington, DC, USA (2008), http://dx.doi.org/10.1109/SP.2008.32

[59] Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6), 1157–1210 (Sep 2010), http://dl.acm.org/citation.cfm?id=1891823.1891830

[60] Cousot, P.: Types as abstract interpretations, invited paper. In: Conference Record of the Twentyfourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 316–331. ACM Press, New York, NY, Paris, France (Jan 1997)

[61] Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 238–252. POPL '77, ACM, New York, NY, USA (1977), http://doi.acm.org/10.1145/512950.512973

[62] Cover, T.M., Thomas, J.A.: Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing). Wiley-Interscience, New York, NY, USA (2006)

[63] Crosby, S.A., Wallach, D.S.: Denial of service via algorithmic complexity attacks. In: Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12. pp. 3–3. SSYM'03, USENIX Association, Berkeley, CA, USA (2003), http://dl.acm.org/citation.cfm?id=1251353.1251356

[64] Davies, K.: kjd (2021), https://github.com/kjd/idan, [Online; accessed 19-July-2021]

[65] Davis, J.C., Michael IV, L.G., Coghlan, C.A., Servant, F., Lee, D.: Why aren't regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 443–454 (2019)

[66] Denning, D.E.: A lattice model of secure information flow. Commun. ACM **19**(5), 236–243 (May 1976), http://doi.acm.org/10.1145/360051.360056

[67] Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM **20**(7), 504–513 (Jul 1977), http://doi.acm.org/10.1145/359636.359712

[68] Denning, D.E.R.: Secure Information Flow in Computer Systems. Ph.D. thesis, Purdue University, West Lafayette, IN, USA (1975), aAI7600514

[69] Devriese, D., Piessens, F.: Noninterference through secure multi-execution. In: Proceedings of the 2010 IEEE Symposium on Security and Privacy. pp. 109–124. SP '10, IEEE Computer Society, Washington, DC, USA (2010), http://dx.doi.org/10.1109/SP.2010.15

[70] Dhar, A., Purandare, R., Dhawan, M., Rangaswamy, S.: Clotho: Saving programs from malformed strings and incorrect string-handling. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. p. 555–566. ESEC/FSE 2015, Association for Computing Machinery, New York, NY, USA (2015), https://doi.org/10.1145/2786805.2786877

[71] Disney, T., Flanagan, C.: Gradual information flow typing (2011)

[72] Disney, T., Flanagan, C.: Gradual information flow typing (2011)

[73] Eghbali, A., Pradel, M.: No strings attached: An empirical study of string-related software bugs. ASE (2020)

[74] Ehrig, H., Mahr, B., Orejas, F.: Introduction to algebraic specification. part 2: From classical view to foundations of system specifications. The Computer Journal **35**(5), 468–477 (1992)

[75] Empson, W.: Collected Poems of William Empson. Harvest book, Harcourt, Brace (1949), https://books.google.nl/books?id=_8QPAAAAMAAJ

[76] Facebook: Flow. Facebook (2017), http://flow.org/

[77] Facebook: The Hack Programming Language. Facebook (2017), http://hacklang.org/

[78] Fennell, L., Thiemann, P.: Gradual security typing with references. In: Proceedings of the 2013 IEEE 26th Computer Security Foundations Symposium. pp. 224–239. CSF '13, IEEE Computer Society, Washington, DC, USA (2013), http://dx.doi.org/10.1109/CSF.2013.22

[79] Fennell, L., Thiemann, P.: LJGS: Gradual Security Types for Object-Oriented Languages. In: Krishnamurthi, S., Lerner, B.S. (eds.) 30th European Conference on Object-Oriented Programming (ECOOP 2016). Leibniz International Proceedings in Informatics (LIPIcs), vol. 56, pp. 1–26. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2016), http://drops.dagstuhl.de/opus/volltexte/2016/6103

[80] Fielding, R.T., Taylor, R.N.: Architectural styles and the design of network-based software architectures, vol. 7. University of California, Irvine Irvine (2000)

[81] Fielding, R.T., Taylor, R.N.: Architectural Styles and the Design of Network-Based Software Architectures. Ph.D. thesis (2000), aAI9980887

[82] Findler, R.B., Felleisen, M.: Contracts for higher-order functions. SIGPLAN Not. **37**(9), 48–59 (Sep 2002), `http://doi.acm.org/10.1145/583852.581484`

[83] Ford, B.: Parsing expression grammars: a recognition-based syntactic foundation. In: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 111–122 (2004)

[84] Fowler, M.: Refactoring - Improving the Design of Existing Code. Addison Wesley object technology series, Addison-Wesley, Reading, Massachusetts (1999), `http://martinfowler.com/books/refactoring.html`

[85] Fraser, G., Arcuri, A.: A large scale evaluation of automated unit test generation using evosuite. ACM Transactions on Software Engineering and Methodology (TOSEM) **24**(2), 8 (2014)

[86] Gabor, D.: Lectures on communication theory. Technical Report (1952)

[87] Ganesh, V., Minnes, M., Solar-Lezama, A., Rinard, M.: What is decidable about strings? (2011)

[88] Gao, W., Kannan, S., Oh, S., Viswanath, P.: Estimating mutual information for discrete-continuous mixtures. In: Proceedings of the 31st Interna-

tional Conference on Neural Information Processing Systems. pp. 5988–5999. NIPS'17, Curran Associates Inc., Red Hook, NY, USA (2017)

[89] Garcia, R., Cimini, M.: Principal type schemes for gradual programs. In: Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 303–315. POPL '15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2676726.2676992

[90] Garcia, R., Clark, A.M., Éric Tanter: Abstracting gradual typing. SIGPLAN Not. **51**(1), 429–442 (Jan 2016), http://doi.acm.org/10.1145/2914770.2837670

[91] Garcia, R., Clark, A.M., Éric Tanter: Abstracting gradual typing. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 429–442. POPL '16, ACM, New York, NY, USA (2016), http://doi.acm.org/10.1145/2837614.2837670

[92] Garcia, R., Tanter, É.: Deriving a simple gradual security language. CoRR **1511.01399** (2015), http://arxiv.org/abs/1511.01399

[93] Garcia, R., Tanter, É.: Deriving a simple gradual security language. CoRR **1511.01399** (2015), http://arxiv.org/abs/1511.01399

[94] Giacobazzi, R., Mastroeni, I.: Abstract non-interference: Parameterizing non-interference by abstract interpretation. SIGPLAN Not. **39**(1), 186–197 (Jan 2004), http://doi.acm.org/10.1145/982962.964017

[95] Goguen, J.A., Meseguer, J.: Security policies and security models. In: IEEE Symposium on Security and Privacy. p. 11. IEEE (1982)

[96] google: atheris (2021), https://github.com/google/atheris, [Online; accessed 19-July-2021]

[97] Groef, W.D., Devriese, D., Nikiforakis, N., Piessens, F.: Flowfox: a web browser with flexible and precise information flow control. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS 2012). pp. 748–759. ACM (2012), https://lirias.kuleuven.be/handle/123456789/354589

[98] Groef, W.D., Devriese, D., Nikiforakis, N., Piessens, F.: Secure multi-execution of web scripts: Theory and practice. J. Comput. Secur. **22**(4), 469–509 (Jul 2014), http://dl.acm.org/citation.cfm?id=2699784.2699786

[99] Gronski, J., Knowles, K., Tomb, A., Freund, S.N., Flanagan, C.: Sage: Hybrid checking for flexible specifications. In: Scheme and Functional Programming Workshop. pp. 93–104 (2006)

[100] Guernic, G.L., Banerjee, A., Jensen, T., Schmidt, D.A.: Automata-based confidentiality monitoring. In: Proceedings of the 11th Asian Computing Science Conference on Advances in Computer Science: Secure Software and Related Issues. pp. 75–89. ASIAN'06, Springer-Verlag, Berlin, Heidelberg (2007), http://dl.acm.org/citation.cfm?id=1782734.1782741

[101] Harman, M., Hu, L., Hierons, R., Wegener, J., Sthamer, H., Baresel, A., Roper, M.: Testability transformation. IEEE Trans. Softw. Eng. **30**(1), 3–16 (Jan 2004), https://doi.org/10.1109/TSE.2004.1265732

[102] Heintze, N., Riecke, J.G.: The SLam calculus: Programming with secrecy and integrity. In: Proceedings of the 25th ACM SIGPLAN-SIGACT Sym-

posium on Principles of Programming Languages. pp. 365–377. POPL '98, ACM, New York, NY, USA (1998), http://doi.acm.org/10.1145/268946.268976

[103] Hennessy, M.: The security picalculus and non-interference (extended abstract). Electron. Notes Theor. Comput. Sci. **83**, 113–129 (Jan 2013), http://dx.doi.org/10.1016/S1571-0661(03)50006-8

[104] Herman, D., Tomb, A., Flanagan, C.: Space-efficient gradual typing. In: In Trends in Functional Programming (TFP) (2007)

[105] Hernández, D.G., Samengo, I.: Estimating the mutual information between two discrete, asymmetric variables with limited samples. Entropy **21**(6) (2019), https://www.mdpi.com/1099-4300/21/6/623

[106] Hicks, B., Ahmadizadeh, K., McDaniel, P.: Understanding practical application development in security-typed languages. In: 22nd Annual Computer Security Applications Conference (ACSAC). Miami, Fl (Dec 2006)

[107] Hosoya, H., Vouillon, J., Pierce, B.C.: Regular expression types for xml. ACM Transactions on Programming Languages and Systems (TOPLAS) **27**(1), 46–90 (2005)

[108] Hunt, S., Sands, D.: On flow-sensitive security types. In: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 79–90. POPL '06, ACM, New York, NY, USA (2006), http://doi.acm.org/10.1145/1111037.1111045

[109] Hutton, G., Meijer, E.: Monadic Parser Combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham (1996)

[110] Igarashi, Y., Sekiyama, T., Igarashi, A.: On polymorphic gradual typing. Proc. ACM Program. Lang. **1**(ICFP), 1–29 (Aug 2017), `http://doi.acm.org/10.1145/3110284`

[111] Ina, L., Igarashi, A.: Gradual typing for generics. SIGPLAN Not. **46**(10), 609–624 (Oct 2011), `http://doi.acm.org/10.1145/2076021.2048114`

[112] Jennings, H.A.: Rosie pattern language (2020), `https://rosie-lang.org/index.html`, [Online; accessed 05-03-2020]

[113] Joshi, R., Leino, K.R.M.: A semantic approach to secure information flow (2000)

[114] King, D., Hicks, B., Hicks, M., Jaeger, T.: Implicit Flows: Can't Live with 'Em, Can't Live without 'Em, pp. 56–70. Springer Berlin Heidelberg, Berlin, Heidelberg (2008), `http://dx.doi.org/10.1007/978-3-540-89862-7{_}4`

[115] King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (Jul 1976), `http://doi.acm.org/10.1145/360248.360252`

[116] Klensin, J.: Internationalized domain names in applications (2021), `https://datatracker.ietf.org/doc/html/rfc5891`, [Online; accessed 19-August-2021]

[117] Kobayashi, N.: Type-based information flow analysis for the π-calculus. Acta Inf. **42**(4), 291–347 (Dec 2005), `http://dx.doi.org/10.1007/s00236-005-0179-x`

[118] Kraskov, A., Stögbauer, H., Grassberger, P.: Estimating mutual information. Phys. Rev. E **69**, 066138 (Jun 2004), https://link.aps.org/doi/10.1103/PhysRevE.69.066138

[119] Kroening, D., Tautschnig, M.: CBMC – C bounded model checker. In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 389–391. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)

[120] Kuhlenschmidt, A., Almahallawi, D., Siek, J.G.: Efficient gradual typing. CoRR **1802.06375** (2018), http://arxiv.org/abs/1802.06375

[121] Kuhlenschmidt, A., Almahallawi, D., Siek, J.G.: Toward efficient gradual typing for structural types via coercions. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 517–532. ACM (2019)

[122] Lehmann, N., Éric Tanter: Gradual refinement types. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. pp. 775–788. POPL 2017, ACM, New York, NY, USA (2017), http://doi.acm.org/10.1145/3009837.3009856

[123] Li, P., Zdancewic, S.: Downgrading policies and relaxed noninterference. SIGPLAN Not. **40**(1), 158–170 (Jan 2005), http://doi.acm.org/10.1145/1047659.1040319

[124] Liang, T., Reynolds, A., Tinelli, C., Barrett, C., Deters, M.: A dpll(t) theory solver for a theory of strings and regular expressions. In: Proceedings of the 16th International Conference on Computer Aided Verification - Volume

8559. pp. 646–662. Springer-Verlag, Berlin, Heidelberg (2014), https://doi.org/10.1007/978-3-319-08867-9_43

[125] LibFuzzer: A library for coverage-guided fuzz testing (2019), https://llvm.org/docs/LibFuzzer.html, [Online; accessed 13-July-2021]

[126] MacIver, D., Hatfield-Dodds, Z., Contributors, M.: Hypothesis: A new approach to property-based testing. Journal of Open Source Software **4**(43), 1891 (Nov 2019), http://dx.doi.org/10.21105/joss.01891

[127] Malecha, G., Chong, S.: A more precise security type system for dynamic security tests. In: Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security. pp. 1–12. PLAS '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1814217.1814221

[128] Marron, M.: Bosque language (2019), https://github.com/microsoft/BosqueLanguage/blob/master/docs/language/overview.md, [Online; accessed 20-11-2019]

[129] Marron, M.: Regularized programming with the bosque language. Tech. Rep. MSR-TR-2019-10, Microsoft (April 2019), https://www.microsoft.com/en-us/research/publication/regularized-programming-with-the-bosque-language/

[130] Masri, W.A.: Dynamic information flow analysis, slicing and profiling. Case Western Reserve University (2005)

[131] Masri, W., Podgurski, A.: Measuring the strength of information flows in programs. ACM Trans. Softw. Eng. Methodol. **19**(2) (Oct 2009), https://doi.org/10.1145/1571629.1571631

[132] McCamant, S., Ernst, M.D.: Quantitative information-flow tracking for C and related languages (2006)

[133] McCamant, S., Ernst, M.D.: Quantitative information flow as network flow capacity. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 193–205 (2008)

[134] McKerns, M.M., Strand, L., Sullivan, T., Fang, A., Aivazis, M.A.G.: Building a framework for predictive science (2012)

[135] Michael, L.G., Donohue, J., Davis, J.C., Lee, D., Servant, F.: Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering. p. 415–426. ASE '19, IEEE Press (2019), https://doi.org/10.1109/ASE.2019.00047

[136] Mitchell, J.: Foundations for Programming Languages. Foundations of computing, MIT Press (1996), https://books.google.co.uk/books?id=KyCLQgAACAAJ

[137] Mu, C., Clark, D.: A tool: quantitative analyser for programs. In: Proceedings of the 8th Conference on Quantitative Evaluation of Systems. pp. 145–146. QEST 2011 (2011)

[138] Mu, C.: Quantitative program dependence graphs. In: Aoki, T., Taguchi, K. (eds.) Formal Methods and Software Engineering. pp. 103–118. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)

[139] Muehlboeck, F., Tate, R.: Sound gradual typing is nominally alive and well. In: OOPSLA. ACM, New York, NY, USA (2017), http://www.cs.cornell.edu/~ross/publications/nomalive/

[140] Myers, A.C., Liskov, B.: Protecting privacy using the decentralized label model. ACM Trans. Softw. Eng. Methodol. **9**(4), 410–442 (Oct 2000), http://doi.acm.org/10.1145/363516.363526

[141] Myers, A.C., Zheng, L., Zdancewic, S., Chong, S., Nystrom, N.: Jif 3.0: Java information flow (Jul 2006), http://www.cs.cornell.edu/jif

[142] Nemenman, I.: Coincidences and estimation of entropies of random variables with large cardinalities. Entropy **13**(12), 2013–2023 (2011)

[143] Nemenman, I., Bialek, W., de Ruyter van Steveninck, R.: Entropy and information in neural spike trains: Progress on the sampling problem. Physical Review E **69**(5), 056111 (2004)

[144] Nemenman, I., Shafee, F., Bialek, W.: Entropy and inference, revisited. In: Advances in neural information processing systems. pp. 471–478 (2002)

[145] New, M.S., Ahmed, A.: Graduality from embedding-projection pairs. Proc. ACM Program. Lang. **2**(ICFP), 1–30 (Jul 2018), http://doi.acm.org/10.1145/3236768

[146] Nicola, R.D., Ferrari, G., Pugliese, R.: Programming Access Control: The Klaim Experience, pp. 48–65. Springer Berlin Heidelberg, Berlin, Heidelberg (2000), http://dx.doi.org/10.1007/3-540-44618-4{_}5

[147] Omar, C., Aldrich, J.: Reasonably programmable literal notation. Proceedings of the ACM on Programming Languages **2**, 1–32 (07 2018)

[148] Omar, C., Kurilova, D., Nistor, L., Chung, B., Potanin, A., Aldrich, J.: Safely composable type-specific languages (2014)

[149] Ou, X., Tan, G., Mandelbaum, Y., Walker, D.: Dynamic Typing with Dependent Types, pp. 437–450. Springer US, Boston, MA (2004), `http://dx.doi.org/10.1007/1-4020-8141-3{_}34`

[150] Paninski, L.: Estimation of entropy and mutual information. Neural Computation **15**, 1191–1253 (2003)

[151] Paninski, L.: Estimation of entropy and mutual information. Neural computation **15**(6), 1191–1253 (2003)

[152] PCRE2 contributors: pcre2 man page (2019), `https://www.pcre.org/current/doc/html/pcre2.html`, [Online; accessed 18-11-2019]

[153] Pierce, B.C.: Types and Programming Languages. Mit Press, MIT Press (2002), `https://books.google.co.uk/books?id=ti6zoAC9Ph8C`

[154] Pierce, B.C.: Types and Programming Languages. The MIT Press, 1st edn. (2002)

[155] Potts, D., Bourquin, R., Andresen, L., Andronick, J., Klein, G., Heiser, G.: Mathematically verified software kernels: Raising the bar for high assurance implementations. Technical report, NICTA, Sydney, Australia (Jul 2014)

[156] Rajani, V., Garg, D.: Types for information flow control: Labeling granularity and semantic models. In: 31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018. pp. 233–246 (2018), `https://doi.org/10.1109/CSF.2018.00024`

[157] Rastogi, A., Swamy, N., Fournet, C., Bierman, G., Vekris, P.: Safe & efficient gradual typing for typescript. SIGPLAN Not. **50**(1), 167–180 (Jan 2015), `http://doi.acm.org/10.1145/2775051.2676971`

[158] Reynolds, J.C.: What do types mean? — From intrinsic to extrinsic semantics, pp. 309–327. Springer New York, New York, NY (2003), https://doi.org/10.1007/978-0-387-21798-7{_}15

[159] Rice, A., Aftandilian, E., Jaspan, C., Johnston, E., Pradel, M., Arroyo-Paredes, Y.: Detecting argument selection defects. Proc. ACM Program. Lang. **1**(OOPSLA), 104:1–104:22 (Oct 2017), http://doi.acm.org/10.1145/3133928

[160] Romanelli, M., Chatzikokolakis, K., Palamidessi, C., Piantanida, P.: Estimating g-leakage via machine learning. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. pp. 697–716 (2020)

[161] Rosenwasser, D.: String literal types. https://github.com/Microsoft/TypeScript/pull/5185 (2015), accessed: 2019-03-06

[162] Russo, A., Sabelfeld, A.: Dynamic vs. static flow-sensitive security analysis. In: Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium. pp. 186–199. CSF '10, IEEE Computer Society, Washington, DC, USA (2010), http://dx.doi.org/10.1109/CSF.2010.20

[163] Russo, A., Sabelfeld, A.: Dynamic vs. static flow-sensitive security analysis. In: 2010 23rd IEEE Computer Security Foundations Symposium. pp. 186–199 (2010)

[164] Russo, A., Sabelfeld, A., Li, K.: Implicit flows in malicious and nonmalicious code. In: Logics and Languages for Reliability and Security, pp. 301–

322. NATO (2010), http://dx.doi.org/10.3233/978-1-60750-100-8-301

[165] Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J.Sel. A. Commun. **21**(1), 5–19 (Sep 2006), https://doi.org/10.1109/JSAC.2002.806121

[166] Sabelfeld, A., Russo, A.: From dynamic to static and back: Riding the roller coaster of information-flow control research. In: Proceedings of the 7th International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics. pp. 352–365. PSI'09, Springer-Verlag, Berlin, Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-11486-1{_}30

[167] Sabelfeld, A., Sands, D.: Declassification: Dimensions and principles. J. Comput. Secur. **17**(5), 517–548 (Oct 2009), http://dl.acm.org/citation.cfm?id=1662658.1662659

[168] nados Schwerter, F.B., Garcia, R., Éric Tanter: A theory of gradual effect systems. SIGPLAN Not. **49**(9), 283–295 (Aug 2014), http://doi.acm.org/10.1145/2692915.2628149

[169] Schwerter, F.B., Garcia, R., Tanter, É.: Gradual type-and-effect systems. J. Funct. Program. **26**, e19 (2016), https://doi.org/10.1017/S0956796816000162

[170] Scott, D.S.: Relating theories of the λ-calculus. In: Hindley, R., Seldin, J. (eds.) To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalisms. pp. 403–450. Academic Press (1980)

[171] Sergey, I., Clarke, D.: Gradual ownership types. In: Proceedings of the 21st European Conference on Programming Languages and Systems. pp.

579–599. ESOP'12, Springer-Verlag, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-28869-2{_}29

[172] Shannon, C.E.: A mathematical theory of communication. The Bell System Technical Journal **27**, 379–423 (Jul 1948), http://cm.bell-labs.com/cm/ms/what/shannonday/shannon1948.pdf

[173] Siek, J., Taha, W.: Gradual typing for objects. In: Proceedings of the 21st European Conference on ECOOP 2007: Object-Oriented Programming. pp. 2–27. ECOOP '07, Springer-Verlag, Berlin, Heidelberg (2007), http://dx.doi.org/10.1007/978-3-540-73589-2{_}2

[174] Siek, J., Taha, W.: Gradual typing for objects. In: European Conference on Object-Oriented Programming. pp. 2–27. Springer (2007)

[175] Siek, J., Thiemann, P., Wadler, P.: Blame and coercion: Together again for the first time. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 425–435. PLDI '15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2737924.2737968

[176] Siek, J.G., Taha, W.: Gradual typing for functional languages. In: IN SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP. pp. 81–92 (2006)

[177] Siek, J.G., Taha, W.: Gradual typing for functional languages. In: IN SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP. pp. 81–92 (2006)

[178] Siek, J.G., Vachharajani, M.: Gradual typing with unification-based inference. In: Proceedings of the 2008 Symposium on Dynamic Languages. pp.

1–12. DLS ’08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1408681.1408688

[179] Siek, J.G., Vitousek, M.M., Cimini, M., Boyland, J.T.: Refined Criteria for Gradual Typing. In: Ball, T., Bodik, R., Krishnamurthi, S., Lerner, B.S., Morrisett, G. (eds.) 1st Summit on Advances in Programming Languages (SNAPL 2015). Leibniz International Proceedings in Informatics (LIPIcs), vol. 32, pp. 274–293. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2015), http://drops.dagstuhl.de/opus/volltexte/2015/5031

[180] Siek, J.G., Vitousek, M.M., Cimini, M., Boyland, J.T.: Refined Criteria for Gradual Typing. In: Ball, T., Bodik, R., Krishnamurthi, S., Lerner, B.S., Morrisett, G. (eds.) 1st Summit on Advances in Programming Languages (SNAPL 2015). Leibniz International Proceedings in Informatics (LIPIcs), vol. 32, pp. 274–293. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2015), http://drops.dagstuhl.de/opus/volltexte/2015/5031

[181] Siek, J.G., Vitousek, M.M., Cimini, M., Tobin-Hochstadt, S., Garcia, R.: Monotonic references for efficient gradual typing. In: Vitek, J. (ed.) Programming Languages and Systems. pp. 432–456. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)

[182] Simonet, V.: The flow caml system. software release (2015), http://www.normalesup.org/~simonet/soft/flowcaml/

[183] Simonet, V.: Flow Caml in a nutshell. In: Hutton, G. (ed.) Proceedings of the first APPSEM-II workshop. pp. 152–165. Nottingham, United Kingdom (Mar 2003)

[184] Sipser, M.: Introduction to the Theory of Computation. International Thomson Publishing, 1st edn. (1996)

[185] Smith, G.: A new type system for secure information flow. In: Proceedings of the 14th IEEE Workshop on Computer Security Foundations. pp. 115–125. CSFW '01, IEEE Computer Society, Washington, DC, USA (2001), http://dl.acm.org/citation.cfm?id=872752.873517

[186] Smith, G.: Principles of Secure Information Flow Analysis, pp. 291–307. Springer US, Boston, MA (2007), http://dx.doi.org/10.1007/978-0-387-44599-1{_}13

[187] Smith, G.: On the foundations of quantitative information flow. In: Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009. Springer-Verlag (2009)

[188] Smith, G.: On the foundations of quantitative information flow. In: de Alfaro, L. (ed.) Foundations of Software Science and Computational Structures. pp. 288–302. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)

[189] Stefan, D., Russo, A., Mitchell, J.C., Mazières, D.: Flexible dynamic information flow control in haskell. SIGPLAN Not. **46**(12), 95–106 (Sep 2011), http://doi.acm.org/10.1145/2096148.2034688

[190] Stefan, D., Russo, A., Mitchell, J.C., Mazières, D.: Flexible dynamic information flow control in the presence of exceptions. CoRR **1207.1457** (2012), http://arxiv.org/abs/1207.1457

[191] Swamy, N., Fournet, C., Rastogi, A., Bhargavan, K., Chen, J., Strub, P.Y., Bierman, G.: Gradual typing embedded securely in javascript. SIGPLAN Not. **49**(1), 425–437 (Jan 2014), http://doi.acm.org/10.1145/2578855.2535889

[192] Swamy, N., Hicks, M., Tse, S., Zdancewic, S.: Managing policy updates in security-typed languages. In: Proceedings of the 19th IEEE Workshop on Computer Security Foundations. pp. 202–216. CSFW '06, IEEE Computer Society, Washington, DC, USA (2006), http://dx.doi.org/10.1109/CSFW.2006.17

[193] Takikawa, A., Feltey, D., Greenman, B., New, M.S., Vitek, J., Felleisen, M.: Is sound gradual typing dead? SIGPLAN Not. **51**(1), 456–468 (Jan 2016), http://doi.acm.org/10.1145/2914770.2837630

[194] Takikawa, A., Feltey, D., Greenman, B., New, M.S., Vitek, J., Felleisen, M.: Is sound gradual typing dead? In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 456–468. POPL '16, ACM, New York, NY, USA (2016), http://doi.acm.org/10.1145/2837614.2837630

[195] Éric Tanter, Tabareau, N.: Gradual certified programming in coq. SIGPLAN Not. **51**(2), 26–40 (Oct 2015), http://doi.acm.org/10.1145/2936313.2816710

[196] development team, T.C.: The Coq proof assistant reference manual. LogiCal Project (2004), http://coq.inria.fr, version 8.0

[197] Team, T.: Advanced types. https://www.typescriptlang.org/docs/handbook/advanced-types.html (2019), [Online; accessed 04-04-2019]

[198] Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In: Proceedings of the 12th International Conference on Static Analysis. pp. 352–367. SAS'05, Springer-Verlag, Berlin, Heidelberg (2005), http://dx.doi.org/10.1007/11547662{_}24

[199] Thatte, S.: Quasi-static typing. In: Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 367–381. POPL '90, ACM, New York, NY, USA (1990), http://doi.acm.org/10.1145/96709.96747

[200] Thiemann, P.: Session types with gradual typing. Symposium on Trustworthy Global Computing. (2014)

[201] tlsfuzzer: python-ecsda (2021), https://github.com/tlsfuzzer/python-ecsda, [Online; accessed 19-August-2021]

[202] Tobin-Hochstadt, S., Felleisen, M.: Interlanguage migration: From scripts to programs. In: Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications. pp. 964–974. OOPSLA '06, ACM, New York, NY, USA (2006), http://doi.acm.org/10.1145/1176617.1176755

[203] Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of typed scheme. SIGPLAN Not. **43**(1), 395–406 (Jan 2008), http://doi.acm.org/10.1145/1328897.1328486

[204] Toro, M., Garcia, R., Éric Tanter: Type-driven gradual security with references. ACM Trans. Program. Lang. Syst. **40**(4), 1–55 (Dec 2018), http://doi.acm.org/10.1145/3229061

[205] TypeScript: Suggestion: Regex-validated string type. https://github.com/Microsoft/TypeScript/issues/6579 (2016), accessed: 2019-03-06

[206] Vallee-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot: A java bytecode optimization framework. In: CASCON First Decade High Impact Papers. pp. 214–224. IBM Corp. (2010)

[207] Vassena, M., Russo, A., Garg, D., Rajani, V., Stefan, D.: From fine- to coarse-grained dynamic information flow control and back. Proc. ACM Program. Lang. **3**(POPL) (Jan 2019), https://doi.org/10.1145/3290389

[208] Vitousek, M.M., Kent, A.M., Siek, J.G., Baker, J.: Design and evaluation of gradual typing for python. SIGPLAN Not. **50**(2), 45–56 (Oct 2014), http://doi.acm.org/10.1145/2775052.2661101

[209] Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. J. Comput. Secur. **4**(2-3), 167–187 (Jan 1996), http://dl.acm.org/citation.cfm?id=353629.353648

[210] Volpano, D., Smith, G.: Probabilistic noninterference in a concurrent language. J. Comput. Secur. **7**(2-3), 231–253 (Mar 1999), http://dl.acm.org/citation.cfm?id=353594.353608

[211] Wadler, P.: The essence of functional programming. In: Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming

Languages. pp. 1–14. POPL ’92, ACM, New York, NY, USA (1992), http://doi.acm.org/10.1145/143165.143169

[212] Wadler, P., Findler, R.B.: Well-typed programs can’t be blamed. In: Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009. pp. 1–16. ESOP ’09, Springer-Verlag, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-00590-9_1

[213] Wheeler, J.A.: Information, physics, quantum: the search for links (1999)

[214] Wiesstein, E.W.: Trigamma function. from mathworld – A wolfram web resource (2021), https://mathworld.wolfram.com/TrigammaFunction.html, [Online; accessed 29-Nov-2021]

[215] Wikipedia contributors: Dyck language — Wikipedia, the free encyclopedia (2020), https://en.wikipedia.org/wiki/Dyck_language, [Online; accessed 14-05-2020]

[216] Wikipedia contributors: Kendall rank correlation coefficient — wikipedia the free encyclopedia (2020), https://en.wikipedia.org/wiki/Kendall{_}rank{_}correlation{_}coefficient, [Online; accessed 15-Feb-2020]

[217] Wikipedia contributors: Galois connection — wikipedia the free encyclopedia (2021), https://en.wikipedia.org/wiki/Galois{_}connection, [Online; accessed 16-June-2021]

[218] Wikipedia contributors: Murmurhash — wikipedia the free encyclopedia (2021), https://en.wikipedia.org/wiki/MurmurHash#cite{_}note-4, [Online; accessed 16-June-2021]

[219] Wikipedia contributors: Wasserstein metric— wikipedia the free encyclopedia (2021), https://en.wikipedia.org/wiki/Wasserstein_metric, [Online; accessed 08-Nov-2021]

[220] Wright, H.K.: Incremental type migration using type algebra. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 756–765 (2020). https://doi.org/10.1109/ICSME46990.2020.00085

[221] Yu, F., Alkhalaf, M., Bultan, T., Ibarra, O.H.: Automata-based symbolic string analysis for vulnerability detection. Form. Methods Syst. Des. **44**(1), 44–70 (Feb 2014), http://dx.doi.org/10.1007/s10703-013-0189-1

[222] Yu, Z., Bai, C., Seinturier, L., Monperrus, M.: Characterizing the usage, evolution and impact of java annotations in practice. IEEE Transactions on Software Engineering (2019)

[223] Zdancewic, S.A.: Programming Languages for Information Security. Ph.D. thesis, Cornell University, Ithaca, NY, USA (2002), aAI3063751

[224] Zdancewic, S.: A type system for robust declassification. Electronic Notes in Theoretical Computer Science **83**(Supplement C), 263–277 (2003), http://www.sciencedirect.com/science/article/pii/S1571066103500147, proceedings of 19th Conference on the Mathematical Foundations of Programming Semantics

[225] Zeng, X., Xia, Y., Tong, H.: Jackknife approach to the estimation of mutual information. Proceedings of the National Academy of Sciences **115**(40), 9956–9961 (2018), https://www.pnas.org/content/115/40/9956

[226] Zheng, Y., Zhang, X., Ganesh, V.: Z3-str: A z3-based string solver for web application analysis. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. pp. 114–124. ESEC/FSE 2013, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2491411.2491456