Scripting languages and frameworks
analysis and verification (Dagstuhl Seminar 14271)

Henglein, Fritz; Jhala, Ranjit; Krishnamurthi, Shriram; Thiemann, Peter

DOI:
10.4230/DagRep.4.6.84

Publication date:
2014

Document version
Publisher's PDF, also known as Version of record

Citation for published version (APA):
Henglein, F., Jhala, R., Krishnamurthi, S., & Thiemann, P. (Eds.) (2014). *Scripting languages and frameworks: analysis and verification (Dagstuhl Seminar 14271)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. Dagstuhl Reports, No. 6, Vol.. 4 https://doi.org/10.4230/DagRep.4.6.84

Report from Dagstuhl Seminar 14271

Scripting Languages and Frameworks: Analysis and Verification

Edited by

Fritz Henglein[1], Ranjit Jhala[2], Shriram Krishnamurthi[3], and Peter Thiemann[4]

1 **University of Copenhagen, DK,** henglein@diku.dk
2 **University of California – San Diego, US,** jhala@cs.ucsd.edu
3 **Brown University – Providence, US,** sk@cs.brown.edu
4 **Universität Freiburg, DE,** thiemann@informatik.uni-freiburg.de

Abstract

This report documents the program and the outcomes of Dagstuhl Seminar 14271 "Scripting Languages and Frameworks: Analysis and Verification". The seminar brought together a broad spectrum of researchers working on the semantics, analysis and verification of scripting languages. In addition to talks describing the latest problems and research on the key issues, split roughly into four overarching themes: semantics, types, analysis, contracts, languages, and security, the seminar had breakout sessions devoted to crosscutting topics that were of broad interest across the community, including, how to create shared analysis infrastructure, how to think about the semantics of contracts and blame, and the role of soundness in analyzing real world languages, as well as several "tutorial" sessions explaining various new tools and techniques.

Seminar July 1–4, 2014 – http://www.dagstuhl.de/14271
1998 ACM Subject Classification D.3.3 Programming Languages, F.3.1 Logics and Meanings of Programs
Keywords and phrases Scripting Languages, Frameworks, Contracts, Types, Analysis, Semantics
Digital Object Identifier 10.4230/DagRep.4.6.84

1 Executive Summary

Fritz Henglein
Ranjit Jhala
Shriram Krishnamurthi
Peter Thiemann

In the past decade scripting languages have become more mature: the wild experimentation and almost wilful embrace of obfuscation by Perl has been replaced by the level-headed simplicity of Python and the embrace of programming language research roots by Ruby. As a result, these languages have moved into the mainstream: every Web user relies on JavaScript.

The Challenges of Scripting Languages Though scripting languages have become more mature, from the perspective of building robust, reliable software, they still suffer from several distinct problems, each of which creates new challenges for the research community.

- While these languages have textual definitions, they lack more formal descriptions, and in practice the textual "definitions" are themselves often in conflict with the normative

DAGSTUHL REPORTS Dagstuhl Reports
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

nature of the implementations. This is in contrast to languages like Standard ML where the formal definition comes first. *How far can we go in creating formal semantics from a combination of implementations and textual documents?*

- Tests – more than either implementations, textual definitions, or formal semantics – are becoming the norm for specification. For instance, the latest JavaScript standard explicitly embraces testing by publishing and regularly updating a conformance suite. Similarly, a team trying to create an alternate implementation of one of these languages may read the definition but what they really aspire to match is the test suite behavior. *How can we support test suites as a new avenue of programming language specification?*
- One of the reasons programmers find these languages enjoyable (initially) is that they offer a variety of "convenient" features, such as overloading. As programs grow, however, understanding the full – and unintended! – behaviors of programs becomes a non-trivial effort. *How can we design semantics and static and dynamic tools that can cope with the heavily understated and overloaded behaviors that make scripting languages attractive?*
- Programmers increasingly do not program in languages but in high-level frameworks built atop them. For instance, though "Ruby" is popular for Web programming, programmers rarely write Web applications directly in Ruby, but rather atop the higher-level Ruby on Rails platform. The result of imposing significantly higher-level interfaces is that they necessitate new reasoning modes. For instance, while the jQuery library is a pure JavaScript program, type-checking jQuery as if it were "merely" JavaScript would produce types that are both unreadably compex and relatively useless. *Can we build custom reasoning at the level of the frameworks, then we can provide views of these frameworks that are consistent with the level at which developers think of them, and can we check that the implementations adhere to these interfaces?*
- These languages and frameworks are themselves not enough. They all reside in an eco-system of a family of other languages and frameworks whose interdependencies are necessary for proper understanding of program execution. For instance, in the client-side Web, JavaScript – which has gotten significant attention from the research community – only runs in response to stimuli, which are obtained from the DOM. In turn, the DOM and JavaScript both depend on the style-sheets written in CSS. But in fact all three of these components – the JavaScript code, the CSS styling, and the DOM events – all depend on one another, because almost any one can trigger or modify the other. *Can we construct suitable abstractions such that each language can meaningfully talk about the others without importing an overwhelming amount of detail?*

This seminar brought together a wide variety of researchers working on the above questions. The seminar was organized into a series of short and long talks on topics related to the above overarching questions, and four breakout sessions focussing on broader questions and challenges. Next, we briefly summarize the talks and sessions. The contributed talks focussed on the following over arching themes – *semantics*, *type systems*, *program analysis*, *contracts*, *languages* and *security*.

2 Table of Contents

Overview of Talks: Contracts

Overview of Talks: Languages

Overview of Talks: Security

Lightning Talks

Breakout Sessions

3 Overview of Talks: Semantics

3.1 Python, the Full Monty

Joe Gibbs Politz (Brown University – US)

Joint work of Krishnamurthi, Shriram; Politz, Joe Gibbs

We present a small-step operational semantics for the Python programming language. We present both a core language for Python, suitable for tools and proofs, and a translation process for converting Python source to this core. We have tested the composition of translation and evaluation of the core for conformance with the primary Python implementation, thereby giving confidence in the fidelity of the semantics. We briefly report on the engineering of these components. Finally, we examine subtle aspects of the language, identifying scope as a pervasive concern that even impacts features that might be considered orthogonal.

3.2 An Executable Formal Semantics of PHP

Daniele Filaretti (Imperial College London, GB)

Joint work of Filaretti, Daniele; Maffeis, Sergio
Main reference D. Filaretti, S. Maffeis, "An Executable Formal Semantics of PHP," in Proc. of the 28th Europ. Conf. Object-Oriented Programming (ECOOP'14), LNCS, Vol. 8586, pp. 567–592, Springer, 2014.
URL http://dx.doi.org/10.1007/978-3-662-44202-9_23
URL https://dfilaretti.files.wordpress.com/2014/02/dagstuhl2014.pdf

We describe the first executable formal semantics of a substantial core of PHP – validated by testing against the Zend Test suite.

3.3 JSCert, a two-pronged approach to JavaScript formalization

Alan Schmitt (INRIA Bretagne Atlantique – Rennes, FR)

Main reference M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, G. Smith, "A Trusted Mechanised JavaScript Specification," in Proc. of the 41st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'14), pp. 87–100, ACM, 2014.
URL http://dx.doi.org/10.1145/2535838.2535876

JSCert is a formalization of JavaScript that aims at being as close as possible to the specification while having an executable component to run against test suites.

4 Overview of Talks: Type Systems

4.1 Progressive Types

Joe Gibbs Politz (Brown University – US)

Joint work of Krishnamurthi, Shriram

As modern type systems grow ever-richer, it can become increasingly onerous for programmers to satisfy them. However, some programs may not require the full power of the type system, while others may wish to obtain these rich guarantees incrementally. In particular, programmers may be willing to exploit the safety checks of the underlying run- time system as a substitute for some static guarantees. Progressive types give programmers this freedom, thus creating a gentler and more flexible environment for using powerful type checkers. In this paper we discuss the idea, motivate it with concrete, real-world scenarios, then show the development of a simple progressive type system and present its (progressive) soundness theorem.

4.2 Safe TypeScript

Panagiotis Vekris (University of California – San Diego, US)

Joint work of Rastogi, Aseem; Swamy, Nikhil; Fournet, Cedric; Bierman, Gavin; Vekris, Panagiotis

Safe TypeScript is a gradual type system built on top of the TypeScript compiler framework that achieves type soundness by means of stricter static typing rules and a runtime mechanism for checks lying on the boundary between static and dynamic types. Safe TypeScript is geared towards efficiency: it uses differential subtyping, whereby only a minimum amount of runtime annotations are applied; and provides an erasure modality, which enables selective deletion of type annotations for type constructs that are meant to be dealt with entirely statically. The implemented Safe TypeScript compiler has been succefully used on hundreds of lines of existing TypeScript code, incurring with a modest overhead on sufficiently annotated input code.

4.3 Confined Gradual Typing

Éric Tanter (University of Chile, CL)

Joint work of Allende, Esteban; Fabry, Johan; Garcia, Ronald; Tanter, Éric
Main reference E. Allende, J. Fabry, R. Garcia, É. Tanter, "Confined Gradual Typing," in Proc. of the 2014 ACM Int'l Conf. on Object Oriented Programming Systems Languages & Applications (OOPSLA'14), pp. 251–270, ACM, 2014; pre-print available from author's webpage.
URL http://dx.doi.org/10.1145/2660193.2660222
URL http://pleiad.dcc.uchile.cl/papers/2014/allendeAl-oopsla2014.pdf

Gradual typing combines static and dynamic typing flexibly and safely in a single programming language. To do so, gradually typed languages implicitly insert casts where needed, to ensure at runtime that typing assumptions are not violated by untyped code. However, the implicit nature of cast insertion, especially on higher-order values, can jeopardize reliability and efficiency: higher-order casts can fail at any time, and are costly to execute. We propose Confined Gradual Typing, which extends gradual typing with two new type qualifiers that let programmers control the flow of values between the typed and the untyped worlds, and thereby trade some flexibility for more reliability and performance. We formally develop two variants of Confined Gradual Typing that capture different flexibility/guarantee tradeoffs. We report on the implementation of Confined Gradual Typing in Gradualtalk, a gradually-typed Smalltalk, which confirms the performance advantage of avoiding unwanted higher-order casts and the low overhead of the approach.

4.4 Typing Scheme to Typing Racket

Sam Tobin-Hochstadt (Indiana University – Bloomington, US)

Joint work of Tobin-Hochstadt, Sam; Takikawa, Asumu; Felleisen, Matthias; Strickland, T. Stephen
Main reference A. Takikawa, T. S. Strickland, C. Dimoulas, S. Tobin-Hochstadt, M. Felleisen, "Gradual typing for first-class classes," in Proc. of the 27th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12), pp. 793–810, ACM, 2012; pre-print available from author's webpage.
URL http://www.ccs.neu.edu/racket/pubs/oopsla12-tsdthf.pdf

We have extended Typed Racket extensively to include support for features that go beyond traditional Scheme, including first-class classes, delimited continuations, mixins, etc.

4.5 Type Systems for JavaScript: Variations on a Theme

Benjamin Lerner (Brown University, US)

Joint work of Lerner, Benjamin; Politz, Joe G.; Guha, Arjun; Krishnamurthi, Shriram
Main reference B. S. Lerner, J. G. Politz, A. Guha, S. Krishnamurthi, "TeJaS: retrofitting type systems for JavaScript," in Proc. of the 9th Symp. on Dynamic Languages (DLS '13), pp. 1–16, ACM, 2013.
URL http://dx.doi.org/10.1145/2508168.2508170

When JavaScript programmers write code, they often target not just the base language but also libraries and API frameworks that drastically change the style of their programs, to the point where they might well be considered as written in domain-specific languages rather than merely JS. Accordingly, the characteristic bugs for such applications varies by domain, and so any tools designed to help developers catch these bugs ought to be tailored to the domain. Yet these tools likely share a common core, since the underlying language is still JS.

We present a TeJaS, a framework for designing type systems for JavaScript that can be customized to analyze the idiomatic errors of various domains, and we illustrate its utility by describing systems for analyzing DOM-access errors in jQuery programs, and privacy violations in Firefox browser extensions running in private-browsing mode.

4.6 Flow Typing

Arjun Guha (University of Massachusetts – Amherst, US)

Joint work of Guha, Arjun; Saftiou, Claudiu; Krishnamurthi, Shriram
Main reference A. Guha, C. Saftoiu, S. Krishnamurthi, "Typing Local Control and State Using Flow Analysis," in Proc. of the 20th Europ. Symp. on Programming (ESOP'11), LNCS, Vol. 6602, pp. 256–275, Springer, 2011.
URL http://dx.doi.org/10.1007/978-3-642-19718-5_14

Programs written in scripting languages employ idioms that confound conventional type systems. In this paper, we highlight one important set of related idioms: the use of local control and state to reason informally about types. To address these idioms, we formalize run-time tags and their relationship to types, and use these to present a novel strategy to integrate typing with flow analysis in a modular way. We demonstrate that in our separation of typing and flow analysis, each component remains conventional, their composition is simple, but the result can handle these idioms better than either one alone.

4.7 Types for Ruby

Jeffrey Foster (University of Maryland, US)

This talk summarizes several years of work on ways to bring some of the benefits of static typing to Ruby. We discuss Diamondback Ruby, a pure static type inference system for Ruby; an extension that does profiling to account for highly dynamic language features; the Mix system, which combines type checking and symbolic execution; and, briefly, RubyDust and rtc, which use the ideas of Mix to provide type inference and checking, respectively, at run time for Ruby.

4.8 Refinement Types for an Imperative Scripting Language

Panagiotis Vekris (University of California – San Diego, US)

Joint work of Jhala, Ranjit

We present a refinement type checker for a scripting language employing various idioms of the JavaScript/TypeScript language family. Our type system consists of a base type system that includes, among others, object types, unions, intersection and higher order functions. On top of this base system lies our refinement type system whose language spans linear arithmetic and uninterpreted predicates. Subtyping on the base system is coercive and the casts added during base typechecking are expressed in the form of refinement type constraints along side value related constraints. These constraints are formulated into logical implications and are discharged by means of Liquid Types inference/checking. Examples outlined in this presentation include safe downcasts based on reflection and in-bounds array accesses.

4.9 Late Typing for Loosely Coupled Recursion

Ravi Chugh (University of California – San Diego, US)

URL http://goto.ucsd.edu/~ravi/research/dagstuhl-late.pptx.pdf

Flexible patterns of mutual recursion can be encoded in scripting languages by defining component functions independently and then "tying the knot" either by mutation through the heap or explicitly passing around receiver objects. We present a mechanism called late typing to reason about such idioms. The key idea is to, first, augment function types with constraints that may not be satisfied when functions are defined and, second, to check that these constraints are satisfied by the time the functions are called.

5 Overview of Talks: Program Analysis

5.1 Abstract Domains for Analyzing Hash Tables

Matthew Might (University of Utah, US)

Hash-table-like abstractions pervade scripting languages as fundamental data structures. (Consider objects in JavaScript, dictionaries in Python and hashes in Ruby.) Attempts to model these abstractions with the same abstract domains used to model abstractions of objects in languages like Java (in which fields and methods are fixed upon allocation) breaks these domains so as to cause catastrophic loss in precision or unsoundness. This talk looks at what is required to retain soundness while more precisely modeling the flexible nature of these structures.

5.2 Static Analysis for Open Objects

Arlen Cox (Colorado University – Boulder, US)

Joint work of Rival, Xavier

In dynamic languages, objects are open – they support iteration over and dynamic addition/deletion of their attributes. Open objects, because they have an unbounded number of attributes, are difficult to abstract without a priori knowledge of all or nearly all of the attributes and thus pose a significant challenge for precise static analysis. To address this challenge, this talks presents the HOO (Heap with Open Objects) abstraction that can precisely represent and infer properties about open-object-manipulating programs without any knowledge of specific attributes. It achieves this by building upon a relational abstract domain for sets that is used to reason about partitions of object attributes. An implementation of the resulting static analysis is used to verify specifications for dynamic language framework code that makes extensive use of open objects, thus demonstrating the effectiveness of this approach.

5.3 Soft Contract Verification

David van Horn (University of Maryland – College Park, US)

Behavioral software contracts are a widely used mechanism for governing the flow of values between components. However, run-time monitoring and enforcement of contracts imposes significant overhead and delays discovery of faulty components to run-time.

To overcome these issues, we present soft contract verification, which aims to statically prove either complete or partial contract correctness of components, written in an untyped, higher-order language with first-class contracts. Our approach uses higher-order symbolic execution, leveraging contracts as a source of symbolic values including unknown behavioral values, and employs an updatable heap of contract invariants to reason about flow-sensitive facts. We prove the symbolic execution soundly approximates the dynamic semantics and that verified programs can't be blamed.

The approach is able to analyze first-class contracts, recursive data structures, unknown functions, and control-flow-sensitive refinements of values, which are all idiomatic in dynamic languages. It makes effective use of an off-the-shelf solver to decide problems without heavy encodings. The approach is competitive with a wide range of existing tools—including type systems, flow analyzers, and model checkers—on their own benchmarks.

5.4 Type Refinement for Static Analysis of JavaScript

Ben Weidermann (Harvey Mudd College, US)

Static analysis of JavaScript has proven useful for a variety of purposes, including optimization, error checking, security auditing, program refactoring, and more. A technique called type refinement that can improve the precision of such static analyses for JavaScript without any discernible performance impact. Refinement is a known technique that uses the conditions in branch guards to refine the analysis information propagated along each branch path. The key insight of this paper is to recognize that JavaScript semantics include many implicit conditional checks on types, and that performing type refinement on these implicit checks provides significant benefit for analysis precision.

5.5 Dynamic Determinacy Analysis

Manu Sridharan (Samsung Research, US)

Joint work of Schäfer, Max; Sridharan, Manu; Dolby, Julian; Tip, Frank
Main reference M. Schäfer, M. Sridharan, J. Dolby, F. Tip, "Dynamic determinacy analysis," in Proc. of the 34th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'13), pp. 165–174, ACM, 2013.
URL http://dx.doi.org/10.1145/2499370.2462168

Programs commonly perform computations that refer only to memory locations that must contain the same value in any program execution. Such memory locations are *determinate* because the value they contain is derived solely from constants. We present a dynamic program analysis that computes a safe approximation of the determinacy of the memory locations referenced at each program point. We implemented this determinacy analysis for JavaScript on top of the `node.js` environment. In two case studies, we demonstrate how the results of determinacy analysis can be used for improving the accuracy of a standard static pointer analysis, and for identifying calls to `eval` that can be eliminated.

5.6 Performance Analysis of JavaScript

Manu Sridharan (Samsung Research, US)

Performance analysis for JavaScript is increasingly important, but difficult due to fragile interactions with JIT compilers and complex native APIs like the DOM. We propose an approach to profiling memory behavior of JavaScript code via heavyweight, platform-independent dynamic tracing and offline analysis, and we outline open challenges with this approach.

5.7 Checking Correctness of TypeScript Interfaces for JavaScript Libraries

Anders Møller (Aarhus University, DK)

Joint work of Møller, Anders; Feldthaus, Asger
Main reference A. Feldthaus, A. Møller, "Checking correctness of TypeScript interfaces for JavaScript libraries," in Proc. of the 2014 ACM Int'l Conf. on Object Oriented Programming Systems Languages & Applications (OOPSLA'14), pp. 1–16, ACM, 2014.
URL http://dx.doi.org/10.1145/2660193.2660215

The TypeScript programming language adds optional types to JavaScript, with support for interaction with existing JavaScript libraries via interface declarations. Such declarations have been written for hundreds of libraries, but they can be difficult to write and often contain errors, which may affect the type checking and misguide code completion for the application code in IDEs.

We present a pragmatic approach to check correctness of TypeScript declaration files with respect to JavaScript library implementations. The key idea in our algorithm is that many declaration errors can be detected by an analysis of the library initialization state combined with a light-weight static analysis of library function code.

Our experimental results demonstrate the effectiveness of the approach: it has found 142 errors in the declaration files of 10 libraries, with an analysis time of a few minutes per library and with a low number of false positives. Our analysis of how programmers use library interface declarations furthermore reveals some practical limitations of the TypeScript type system.

5.8 Analyzing JavaScript Web Applications in the Wild (Mostly) Statically

Sukyoung Ryu (KAIST – Daejeon, KR)

Analyzing real-world JavaScript web applications is a challenging task. On top of understanding the semantics of JavaScript, it requires modeling of web documents, platform objects, and interactions between them. Not only JavaScript itself but also its usage patterns are extremely dynamic. Most of web applications load JavaScript code dynamically, which makes pure static analysis approaches inapplicable. We present our attempts to analyze JavaScript web applications in the wild mostly statically using various approaches to analyze libraries.

6 Overview of Talks: Contracts

6.1 Membranes as Ownership Boundaries

Tom Van Cutsem (Alcatel-Lucent Bell Labs – Antwerp, BE)

We discuss the similarities and differences between membranes and higher-order contracts, give a brief overview of proxies in JS (which are the basic building block for membranes) and then show how membranes can be used to express the use cases typically expressed using ownership type systems.

6.2 TreatJS: Higher-Order Contracts for JavaScript

Matthias Keil (Universität Freiburg, DE)

Joint work of Keil, Matthias; Thiemann, Peter
URL http://www2.informatik.uni-freiburg.de/~keilr/talks/talk_dagstuhl2014-treatjs.pdf

TreatJS is a language embedded, dynamic, higher-order contract system for JavaScript. Beyond the standard abstractions for building higher-order contracts (base, function, and object contracts), TreatJS' novel contribution is its support for boolean combinations of contracts and for the creation of parameterized contracts, which are the building blocks for dependent contracts and more generally run-time generated contracts.

TreatJS is implemented using JavaScript proxies to guarantee full interposition for contracts and it exploits JavaScript's reflective features to run contracts in a sandbox environment. This sandbox guarantees that contracts do not interfere with normal program execution. It also facilitates that all aspects of a contract are specified using the full JavaScript language. No source code transformation or change in the JavaScript run-time system is required.

TreatJS including sandboxing, is formalized and the impact of contracts on execution speed is evaluated in terms of the Google Octane benchmark.

6.3 Contracts for Domain-Specific Languages in Ruby

Jeffrey Foster (University of Maryland, US)

Joint work of Foster, Jeffrey; Strickland, T. Stephen; Ren, Bree

This talk concerns object-oriented embedded DSLs, which are popular in the Ruby community but have received little attention in the research literature. Ruby DSLs implement language keywords as implicit method calls to self; language structure is enforced by adjusting which object is bound to self in different scopes. We propose RDL, a new contract checking system that can enforce contracts on the structure of Ruby DSLs, attributing blame appropriately. We describe RDL and RDLInfer, a tool that infers RDL contracts for existing Ruby DSLs.

7 Overview of Talks: Languages

7.1 HOP: A Multi-tier Language For Web Applications

Tamara Rezk (INRIA Sophia-Antipolis, FR)

We present HOP a multi-tier language to write web applications. We propose a small-step operational semantics to support formal reasoning in HOP. The semantics covers both server side and client side computations, as well as their interactions, and includes creation of web services, distributed client-server communications, concurrent evaluation of service requests at server side, elaboration of HTML documents, DOM operations, evaluation of script nodes in HTML documents and actions from HTML pages at client side.

7.2 Perl: The Ugly Parts

Matthew Might (University of Utah, US)

Let there be no mistake: Perl is extremely useful. Every programmer needs Perl in their arsenal. Thanks to many implicit behaviors, some complex programs can be specified with alarming brevity. Perl excels at extracting and transforming data. But, Perl is as dangerous as it is ugly. This talk looks at the ugly.

7.3 So, What About Lua?

Roberto Ierusalimschy (Pontifical University – Rio de Janeiro, BR)

Lua is a programming language developed at the Catholic University in Rio de Janeiro that came to be the leading scripting language in video games. Lua is also used extensively in embedded devices, such as set-top boxes and TVs, and other applications like Adobe Lightroom and Wikipedia. This talk presents a quick overview of some unconventional aspects of the language.

7.4 Regular Expression Parsing

Bjorn Bugge Grathwohl (University of Copenhagen – DK)

Joint work of Henglein, Fritz and Terp-Rasmussen, Ulrik

Regular expressions (REs) are usually interpreted as languages. For many programming tasks, this is an inadequate interpretation, as it only provides the programmer with a means for testing language membership. Facilities for submatch extraction in tools such as sed and Perl-style REs have been developed to let programmers do data extraction and manipulation with REs. However, the submatch extraction approach is severely limited in its expressibility, as it only allows for a fixed number of submatches, independent of the input size.

Instead, we interpret REs as types. Testing language membership is replaced by a parsing problem: Given an RE E and string s, produce the value (parse tree) in the type T(E) whose flattening is s. With this interpretation, data extraction and manipulation can be performed by writing functional programs that operate on the data types represented by the REs.

We present two automata-based algorithms producing the greedy leftmost parse tree: The two-pass algorithm requires one pass over the input data and an extra pass over an auxiliary data structure; the streaming algorithm implements an optimally streaming parser, in the sense that as soon as the input read so far determines a prefix of all possible parse trees, this prefix is output. This is guaranteed given a PSPACE-complete analysis of the automaton, which can be performed independently of any input strings. However, we conjecture that for "realistic", non-pathological, REs, this analysis is not needed.

7.5 HTML5 Parser Specification and Automated Test Generation

Yasuhiko Minamide (University of Tsukuba, JP)

Joint work of Minamide, Yasuhiko; Mori, Shunsuke
Main reference Y. Minamide, S. Mori, "Reachability Analysis of the HTML5 Parser Specification and Its Application to Compatibility Testing," in Proc. the 18th Int'l Symp. on Formal Methods (FM'12), LNCS, Vol. 7436, pp. 293–307, 2012.
URL http://dx.doi.org/10.1007/978-3-642-32759-9_26

The HTML5 specification includes the detailed specification of the parsing algorithm for HTML5 documents, including error handling. We develop a reachability analyzer for the parsing specification of HTML5 and automatically generate HTML documents to test compatibilities of Web browsers. The set of HTML documents are extracted using our reachability analysis of the statements in the specification. In our preliminary experiments, we generated 353 HTML documents automatically from a subset of the specification and found several compatibility problems by supplying them to Web browsers.

7.6 AmbientTalk: a scripting language for mobile phones

Tom Van Cutsem (Alcatel-Lucent Bell Labs – Antwerp, BE)

We introduce the AmbientTalk programming language, which was designed to script collaborative distributed applications on mobile phones. We give an overview of the language's features and historical roots. We discuss how AmbientTalk is embedded on the JVM, with particular attention to maintaining concurrency invariants.

7.7 Glue Languages

Arjun Guha (University of Massachusetts – Amherst, US)

Joint work of Guha, Arjun; Gupta, Nimish

Puppet is a configuration management system used by thousands of organizations to manage thousands of machines. It is designed to automate tasks such as application configuration, service orchestration, VM provisioning, and more. The heart of Puppet is a declarative domain specific language that, to a first approximation, specifies a collection of resources (e. g., packages, user accounts, files, etc.) to install and the dependencies between them.

Although Puppet performs some static checking, there are many opportunities for errors to occur in Puppet configurations. These errors are very difficult to detect and debug. Even if a configuration is itself bug-free, when a machine is upgraded to a new configuration, it is easy for the machine state and its specified configuration in Puppet to be inconsistent.

8 Overview of Talks: Security

8.1 Information Flow Control in WebKit's JavaScript Bytecode

Christian Hammer (Universität des Saarlandes, DE)

Joint work of Bichhawat, Abhishek; Rajani, Vineet; Garg, Deepak; Hammer, Christian
Main reference A. Bichhawat, V. Rajani, D. Garg, C. Hammer, "Information Flow Control in WebKit's JavaScript Bytecode," in Proc. of the 3rd Int'l Conf. on Principles of Security and Trust (POST'14), LNCS, Vol. 8414, pp. 159–178, Springer, 2014.
URL http://dx.doi.org/10.1007/978-3-642-54792-8_9

Websites today routinely combine JavaScript from multiple sources, both trusted and untrusted. Hence, JavaScript security is of paramount importance. A specific interesting problem is information flow control (IFC) for JavaScript. In this paper, we develop, formalize and implement a dynamic IFC mechanism for the JavaScript engine of a production Web browser (specifically, Safari's WebKit engine). Our IFC mechanism works at the level of JavaScript bytecode and hence leverages years of industrial effort on optimizing both the source to bytecode compiler and the bytecode interpreter. We track both explicit and implicit flows and observe only moderate overhead. Working with bytecode results in new challenges including the extensive use of unstructured control flow in bytecode (which complicates lowering of program context taints), unstructured exceptions (which complicate the matter further) and the need to make IFC analysis permissive. We explain how we address these challenges, formally model the JavaScript bytecode semantics and our instrumentation, prove the standard property of termination-insensitive non-interference, and present experimental results on an optimized prototype.

8.2 Hybrid Information Flow monitoring against Web tracking

Thomas Jensen (INRIA Bretagne Atlantique – Rennes, FR)

Joint work of Bielova, Nataliia; Besson, Frederic; Jensen, Thomas

Motivated by the problem of stateless web tracking (fingerprinting), we propose a novel approach to hybrid information flow monitoring by tracking the knowledge about secret variables using logical formulae. This knowledge representation helps to compare and improve precision of hybrid information flow monitors.

We define a generic hybrid monitor parametrised by a static analysis and derive sufficient conditions on the static analysis for soundness and relative precision of hybrid monitors.

We instantiate the generic monitor with a combined static constant and dependency analysis. Several other hybrid monitors including those based on well-known hybrid techniques for information flow control are formalised as instances of our generic hybrid monitor. These monitors are organised into a hierarchy that establishes their relative precision. The whole framework is accompanied by a formalisation of the theory in the Coq proof assistant.

8.3 Intrusion Detection by Control Flow Analysis

Arjun Guha (University of Massachusetts – Amherst, US)

Joint work of Guha, Arjun; Krishnamurthi, Shriram; Jim, Trevor
Main reference A. Guha, S. Krishnamurthi, T. Jim, “Using Static Analysis for Ajax Intrusion Detection,” in Proc. of the 18th Int’l Conf. on World Wide Web (WWW’09), pp. 561–570, ACM, 2009.
URL http://dx.doi.org/10.1145/1526709.1526785

We present a static control-flow analysis for JavaScript pro- grams running in a web browser. Our analysis tackles numerous challenges posed by modern web applications including asynchronous communication, frameworks, and dynamic code generation. We use our analysis to extract a model of expected client behavior as seen from the server, and build an intrusion-prevention proxy for the server: the proxy intercepts client requests and disables those that do not meet the expected behavior. We insert random asynchronous requests to foil mimicry attacks. Finally, we evaluate our technique against several real applications and show that it protects against an attack in a widely-used web application.

8.4 Multiple Facets for Dynamic Information Flow

Cormac Flanagan (University of California – Santa Cruz, US)

Joint work of Flanagan, Cormac; Austin, Thomas H.
Main reference T. H. Austin, C. Flanagan, “Multiple facets for dynamic information flow,” in Proc. of the 39th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL’12), pp. 165–178, ACM, 2012; pre-print available from author’s webpage.
URL http://dx.doi.org/10.1145/2103656.2103677
URL http://users.soe.ucsc.edu/~cormac/papers/popl12b.pdf

JavaScript has become a central technology of the web, but it is also the source of many security problems, including cross-site scripting attacks and malicious advertising code. Central to these problems is the fact that code from untrusted sources runs with full privileges. We implement information flow controls in Firefox to help prevent violations of data confidentiality and integrity. Most previous information flow techniques have primarily relied on either static type systems, which are a poor fit for JavaScript, or on dynamic analyses that sometimes get stuck due to problematic implicit flows, even in situations where the target web application correctly satisfies the desired security policy. We introduce faceted values, a new mechanism for providing information flow security in a dynamic manner that overcomes these limitations. Taking inspiration from secure multi-execution, we use faceted values to simultaneously and efficiently simulate multiple executions for different security levels, thus providing non-interference with minimal overhead, and without the reliance on the stuck executions of prior dynamic approaches.

8.5 Shill: shell scripting with least authority

Christos Dimoulas (Harvard University, US)

Joint work of Moore, Scott; Dimoulas, Christos; King, Dan; Chong, Stephen

The Principle of Least Authority suggests that software should be executed with no more authority than it requires to accomplish its task. Current security tools make it difficult to apply this principle: they either require significant modifications to applications or do not facilitate reasoning about combining untrustworthy components.

We propose Shill, a secure shell scripting language. Shill scripts enable compositional reasoning about security through declarative security policies that limit the effects of script execution, including the effects of programs invoked by the script. These security policies are a form of documentation for consumers of Shill scripts, and are enforced by the Shill execution environment.

We have implemented a prototype of Shill for FreeBSD. Our evaluation indicates that Shill is a practical and useful system security tool, and can provide fine-grained security guarantees.

8.6 Hybrid Information Flow Analysis for JavaScript

Tamara Rezk (INRIA Sophia-Antipolis, FR)

We propose a novel type system for securing information flow in JavaScript that takes into account the defining features of the language, such as prototypical inheritance, extensible objects, and constructs that check the existence of object properties. The type system infers a set of assertions under which a program can be securely accepted and instruments it so as to dynamically check whether these assertions hold. By deferring rejection to run-time, the hybrid version can typecheck secure programs that purely static type systems cannot accept.

8.7 A Collection of Real World (JavaScript) Security Problems:

Achim D. Brucker (SAP Research – Karlsruhe, DE)

URL http://www.brucker.ch/bibliography/abstract/talk-brucker-js-challenges-2014.en.html

JavaScript is gaining more and more popularity as an implementation language for various applications types such as Web applications (client-side), mobile applications, or server-side applications.

We outline a few security challenges that need to be prevented in such applications and, thus, for which there is a demand for analysis methods that help to detect them during during development.

9 Lightning Talks

9.1 Reasoning about membranes using separation logic

Gareth Smith (Imperial College – UK)

URL http://www.dagstuhl.de/mat/Files/14/14271/14271.SmithGareth.Other.pdf

We propose an extension to separation logic which would make it possible to statically prove security properties of an implementation of a *membrane* program.

9.2 Complexity Analysis of Regular Expression Matching Based on Backtracking

Yasuhiko Minamide (University of Tsukuba, JP)

Joint work of Sugiyama Satoshi; Minamide, Yasuhiko
Main reference S. Sugiyama, Y. Minamide, "Checking Time Linearity of Regular Expression Matching Based on Backtracking," to appear in IPSJ Transactions on Programming.

Regular expression matching is implemented with backtracking in most programming languages. Its time complexity is exponential on the length of a string in worst case. This high complexity causes significant problems in practice. It causes DoS vulnerabilities in server-side applications. It may also affect the result of matching in some implementation with a limit on the number steps in matching, e. g. PCRE. We present a decision procedure to check whether for a given regular expression matching based on backtracking runs in linear time.

9.3 PHPEnkoder: a Wordpress Plugin

Michael Greenberg (Princeton University, US)

URL http://wordpress.org/plugins/php-enkoder/

PHPEnkoder encodes mailto: links and e-mail addresses with JavaScript to stifle webcrawlers. It works by automatically turning plaintext e-mails into (enkoded) links.

Interesting facts:

- Wordpress plugins are installed by being placed in a directory; the files are run at the top level.
- Wordpress plugins are automatically released by tagging in subversion.
- PHPEnkoder parses the page with regular expressions, since Wordpress 'hooks' don't give PHPEnkoder an AST to process, just text.
- Wordpress has an extremely stable API.

For more on this plugin, see http://www.weaselhat.com/phpenkoder/.

9.4 SAST for JavaScript: A Brief Overview of Commercial Tools

Achim D. Brucker (SAP Research – Karlsruhe, DE)

URL http://www.brucker.ch/bibliography/abstract/talk-brucker-sast-js-2014.en.html

Static application security testing (SAST) is a widely used technique that helps to find security vulnerabilities in program code at an early stage in the software development lifecycle. Since a few years, JavaScript is gaining more and more popularity as an implementation language for large applications. Consequently, there is a demand for SAST tools that support JavaScript.

We report briefly on our method for evaluating SAST tools for JavaScript as well as summarize the results of our analysis.

References

1 Achim D. Brucker and Uwe Sodan. Deploying static application security testing on a large scale. In Stefan Katzenbeisser, Volkmar Lotz, and Edgar Weippl, editors, *GI Sicherheit 2014*, volume 228 of *Lecture Notes in Informatics*, pages 91–101. GI, March 2014.

10 Breakout Sessions

In addition to the contributed talks, the seminar had four breakout sessions focussing on cross-cutting issues deemed important by the participants.

10.1 Contracts and Blame

Cormac Flanagan

We discussed some of the counter-intuitive ways in which contracts can fail in systems with multiple modules, and the ways in which blame may be assigned in a manner that may not point at the component that is truly at fault.

10.2 On the Role of Soundness

Matthew Might, Jeffrey Foster

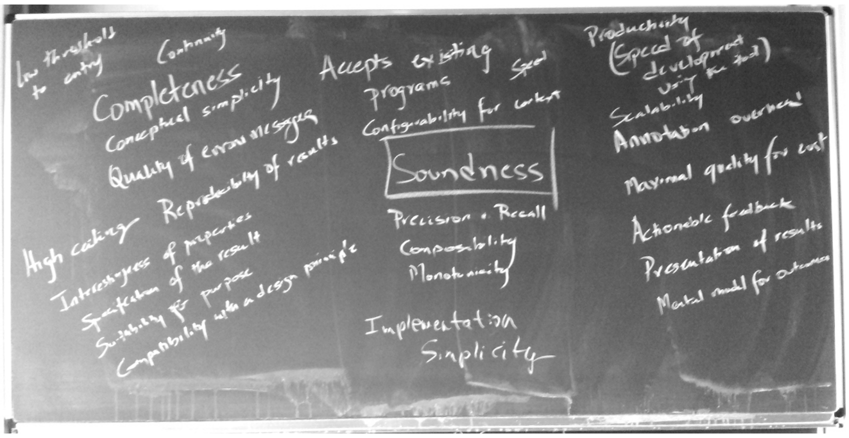
We debated the merits and importance of soundness of tools and analyses for scripting languages. On the one hand, while soundness is essential for relying upon the results of the analysis, on the other, some constructs may be pathologically hard to analyze soundly and even unsound tools may provide extremely invaluable feedback to the developer.

10.3 Metrics for Programming Tools

Krishnamurthi, Shriram; Politz, Joe Gibbs

URL https://drive.google.com/file/d/0B32bNEogmncORS1sN0YtaXZ3V1k/edit?usp=sharing

We gathered metrics for measuring the utility of programming language tools (focused on scripting language applications), prompted by considering alternatives and complements to soundness. See sketch on the blackboard below.


Low thresholds to entry
Continuity
Completeness
Conceptual simplicity
Quality of error messages
Reproducibility of results
High ceiling
Interestingness of properties
Specification of the result
Suitability for purpose
Compatibility with a design principle
Accepts existing programs
Speed
Configurability for context
Soundness
Precision + Recall
Composability
Monotonicity
Implementation Simplicity
Productivity (Speed of development using the tool)
Scalability
Annotation overhead
Maximal quality for cost
Actionable feedback
Presentation of results
Mental model for outcomes

10.4 JavaScript Analysis and Intermediate Representation

Thomas Jensen (INRIA Bretagne Atlantique – Rennes, FR)

Joint work of Jensen, Thomas; Sridharan, Manu

Two issues were discussed:
- how to share models of libraries,
- can we come up with a common intermediate representation for JS analyzers.

The overall goal is to support a re-usable, shared effort. Modeling libraries is not very publishable, hence the need for a collective effort. Another issue is that different kind of models are needed, depending on the analysis. Nevertheless, it was deemed worth to have a common starting point. Models could be written in JS or in an IR or in a formalism that allows integrating elements of abstract domains. One point of view was that it would be valuable to have models satisfying that everything is translatable to the IR, so that different library models can co-exist.

Concerning the IR, several points were discussed:
- Should it accommodate `pre/post` annotations to model libraries?
- Should it be executable (could enable re-injecting into JS to do dynamic analysis)? There is a certain amount of common structure in existing IR so why not just pick one of those.

Some shortcomings were discussed: WALA: not serializable, which is necessary, S5 : should be OK, can be ANF-ed and CPS-ed, MSR IR: has existing formats but prepared to do a clean slate Two different kind of formats were identified: a CFG or something close to the AST. Perhaps there is a need for a series of IR that end in the common format but maximum two seems reasonable to standardize. The discussion ended with a presentation of a proposal for a common IR. The current version can be found at the URL above.

Participants

- Achim D. Brucker
SAP Research – Karlsruhe, DE
- Niels Bjoern Bugge Grathwohl
University of Copenhagen, DK
- Ravi Chugh
University of California – San Diego, US
- Arlen Cox
Univ. of Colorado – Boulder, US
- Christos Dimoulas
Harvard University, US
- Julian Dolby
IBM TJ Watson Research Center – Hawthorne, US
- Matthias Felleisen
Northeastern University – Boston, US
- Daniele Filaretti
Imperial College London, GB
- Cormac Flanagan
University of California – Santa Cruz, US
- Jeffrey Foster
University of Maryland – College Park, US
- Ronald Garcia
University of British Columbia – Vancouver, CA
- Philippa Gardner
Imperial College London, GB
- Michael Greenberg
Princeton University, US
- Arjun Guha
University of Massachusetts – Amherst, US
- Shu-Yu Guo
MOZILLA – Mountain View, US
- Christian Hammer
Universität des Saarlandes, DE
- Fritz Henglein
University of Copenhagen, DK
- Roberto Ierusalimschy
PUC – Rio de Janeiro, BR
- Thomas Jensen
INRIA Bretagne Atlantique – Rennes, FR
- Ranjit Jhala
University of California – San Diego, US
- Matthias Keil
Universität Freiburg, DE
- Shriram Krishnamurthi
Brown University, US
- Benjamin Lerner
Brown University, US
- Benjamin Livshits
Microsoft Res. – Redmond, US
- Sergio Maffeis
Imperial College London, GB
- Matt Might
University of Utah, US
- Yasuhiko Minamide
University of Tsukuba, JP
- Anders Møller
Aarhus University, DK
- Joe Gibbs Politz
Brown University, US
- Ulrik Terp Rasmussen
University of Copenhagen, DK
- Tamara Rezk
INRIA Sophia Antipolis – Méditerranée, FR
- Tiark Rompf
EPFL – Lausanne, CH
- Sukyoung Ryu
KAIST – Daejeon, KR
- Alan Schmitt
INRIA Bretagne Atlantique – Rennes, FR
- Jeremy G. Siek
Univ. of Colorado – Boulder, US
- Gareth Smith
Imperial College London, GB
- Manu Sridharan
Samsung Research, US
- Éric Tanter
University of Chile, CL
- Peter Thiemann
Universität Freiburg, DE
- Sam Tobin-Hochstadt
Indiana University – Bloomington, US
- Tom Van Cutsem
Alcatel-Lucent Bell Labs – Antwerp, BE
- David Van Horn
University of Maryland – College Park, US
- Panagiotis Vekris
University of California – San Diego, US
- Ben Wiedermann
Harvey Mudd College – Claremont, US
- Kwangkeun Yi
Seoul National University, KR