Research Statement

Arjun Guha—University of Massachusetts Amherst

Overview

We are living in an era where ordinary programmers can manipulate their computing infrastructure by just writing code. Instead of asking users to carefully install and configure dependencies, an application can package its entire runtime environment in a container. Instead of buying and maintaining new physical machines, a virtual machine can be deployed in minutes in a datacenter. Even in the network, black-box networking equipment is being replaced by software-defined networks (SDN) that can be reprogrammed using open protocols.

Unfortunately, the languages and abstractions that we use to program infrastructure were designed for an earlier era. This mismatch makes our computing infrastructure brittle and insecure. For example, many network outages are caused by bugs in low-level network policies; many system failures are caused by updates that go awry; and many security vulnerabilities are caused by software that is simply mis-configured.

My research focuses on developing abstractions and verification techniques for modern, configurable computing infrastructure. For the past few years, I've worked on new programming languages and runtime systems for software-defined networking, which is a development that is revolutionizing networking. More recently, I've started working on automated verification techniques for system configuration languages, which have become ubiquitous with the growth of datacenters and cloud computing. In earlier work during my PhD, I developed tools and techniques to verify properties of JavaScript, which is the programming language of the Web.

Network Programming Languages Computer networks do not simply connect machines together, but run several applications on network devices, such as load balancers, intrusion detection systems, authentication portals, and more. Historically, these applications were black-boxes running on proprietary hardware, but software-defined networking now allows anyone to program network devices using open protocols. My research in this area is guided by the question, *What are the right abstractions for programming networks?* Over the past few years, I've made several contributions in this area. I worked on a core calculus for network programming that enables equational reasoning about program transformations and program correctness [1]. I developed high-performance compilers for network programming languages that leverage new data structures and automata theory [28]. I designed a platform that allows applications to safely reconfigure the network to optimize for changing workloads and evolving threats [6]. I developed a certified network control platform that provably eliminates several classes of errors [9].

System Configuration Languages Large modern datacenters and cloud computing has made system configuration a surprisingly challenging problem. In the past few years, several large-scale outages at Facebook[1], Skype[2], the New York Stock Exchange[3], and other organizations have been attributed not to software bugs, but to mis-configured software. At the scale at which these companies operate, system administrators simply cannot manage computing infrastructure manually. They need higher-level abstractions and automated tools to help them specify and verify the state of their systems. In recent work [3], I've started to investigate the question, *What are the right abstractions for managing system configurations?* I've focused on Puppet, which is a system configuration language that is used by hundreds of organizations. Although Puppet provides some rudimentary error-checking, I've shown that serious errors do occur in practice. I've developed a Puppet static analysis that eliminates a large class of errors that are very hard to detect with tests. I've also identified consistency errors that occur when Puppet configurations are updated and developed a program synthesis technique to generate safe configuration updates.

[1] www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919
[2] blogs.skype.com/2015/09/22/skype-outage-an-update-and-an-apology/
[3] www.bloomberg.com/news/articles/2015-07-09/nyse-sec-said-to-suspect-software-update-triggered-trading-halt

JavaScript For my PhD dissertation, I worked on verifying security properties of web programs, which are universally written in JavaScript. A distinguishing characteristic of modern web programs is that they freely run code from third-party sources to display interactive ads, social widgets, user-scripts, and more. However, JavaScript (the programming language of the Web) does not provide any abstractions to isolate modules that should be independent. Instead, as summarized by a recent U.S. Senate report, major websites are routinely compromised by malicious, third-party code [14]. A solution to this problem requires foundational research on Web programming and security, which was the topic of my PhD research. At the time, JavaScript was a very poorly understood language, so I developed an operational semantics for the language and argued that the only way to validate it was to test it against browsers [10]. Since modularity is a key issue with JavaScript and type-checking is fundamentally about modular reasoning, I developed a type-checker for JavaScript that tackles some of its most peculiar features [24, 11]. Finally, I developed a methodology for verifying "Web sandboxes", which are systems that provide abstractions for embedding JavaScript safely. Unsurprisingly, this rigorous formal approach exposed security vulnerabilities in an existing Web sandbox [25, 23]. Several other researchers have since leveraged my work in theirs [2, 8, 18, 27, 15, 17].

The rest of this document discusses my recent research on network programming and system configuration languages, and then concludes with some of my plans for future work. All my work is done in collaboration with colleagues and students who are identified on my C.V.

Programming Networks

Networks are built with a variety of inter-connected devices that implement several in-network features. Routers and switches connect machines together, firewalls and intrusion detection systems filter malicious traffic, load-balancers and caches improve performance, and so on. Traditionally, each device was a proprietary, fixed-function black-box. If a new feature were desired, expensive new hardware would have to be bought. In recent years, this model has been disrupted by software-defined networking, which replaces proprietary, fixed-function equipment with "switches" that can be reprogrammed using open protocols.

SDN protocols, such as OpenFlow, are akin to an assembly language for programmable switches, so researchers have argued for high-level *network programming languages* since the advent of software-defined networking [13]. Like any other high-level language, a network programming language should provide abstractions that hide low-level hardware details from the programmer. Moreover, programmers should be able to reason about network behavior within the language, and should not have to resort to reasoning about low-level details.

Foundations Many network programming languages have powerful programming abstractions, but do not specify the structure of the network [7, 29, 19, 20, 9, 21, 30, 6]. However, the operation of a network is determined both by the programs running on networked devices and the network topology. To answer any questions about whole network behavior, such as *"Can host A communicate with host B?"*, *"Does all traffic traverse an intrusion detection system?"*, or *"Is there a loop in the network?"*, the programmer has to step outside the abstractions of the language.

Our work on NetKAT attempts to resolve this problem [1]. NetKAT is a syntactic theory of network forwarding that supports equational reasoning about network-wide behavior. Using NetKAT, a programmer can answer the kinds of questions listed above and, more generally, reason about equivalences between networks. NetKAT has only a small set of axioms that are sound and complete with respect to its denotational semantics. Despite having a minimal set of primitives, it is straightforward to build higher-level abstractions and prove them correct using the equational theory. Our work shows how to do so for "slices", which are a lightweight form of network virtualization.

The equational theory of NetKAT incorporates Kleene algebra—the algebra of regular expressions—to model sets of paths through the network and Boolean algebra to model how switches filter traffic. Kleene algebra and Boolean algebra had already been synthesized into a unified mathematical structure, called *Kleene Algebra with Tests* (KAT) [16]. The NetKAT equational theory also includes a set of "packet equations" that

describe how packet-testing predicates and packet-modifying primitives relate to each other. The benefit of structuring NetKAT in this manner is that theorems and reasoning techniques for KAT can be applied to NetKAT programs too, even though KAT was originally developed with other applications in mind.

Compilation To be practical, high-level languages need good compilers. Specifically, network compilers need to be fast because they run in a tight loop that responds to network events. For example, when a new host connects to a network, the network program has to be recompiled to route traffic for the host; when a link fails, the network program has to be recompiled to route around the failure; and when traffic patterns shift, the compiler may run again to balance the load. A slow compiler limits how quickly the network can react to changing conditions.

Before our work, the state of the art was to compile network programs directly to tables of packet-processing rules, which is the primitive abstraction provided by networking hardware. This approach, which was pioneered by NetCore [19], was simple and intuitive and can even be proven correct using the equational theory of NetKAT [1]. Unfortunately, this approach is very slow and does not scale to even moderately sized network programs.

In recent work, we developed a compiler for NetKAT that is two orders of magnitude faster than existing systems on a diverse set of benchmarks [28]. The key insight is a new intermediate representation based on a variant of binary decision diagrams that can represent network programs compactly and supports fast, algebraic manipulation. Our new compiler is a hundred times faster than a heavily-optimized version of Pyretic [20] on benchmarks from a recent paper [12]. In fact, the developers of Pyretic have started to integrate our new compiler into their toolchain.

In addition to speed, the new intermediate representation lets us build two powerful new abstractions for network programming. (1) We can compile programs that denote sets of end-to-end paths through the network by automatically inserting stateful operations (e.g., VLAN tagging) to distinguish overlapping paths from each other. To do so, we leverage symbolic automata that use our intermediate representation to encode their transition relation symbolically. (2) We can compile programs written for virtual networks to run on any physical network, with minimal "compatibility requirements" between the virtual and physical topologies. This abstraction models packets as values with two locations—virtual and physical—and synthesizes a program that moves packets along paths in a physical network to account for all possible actions in the virtual network.

Certified Systems NetKAT and other network programming languages abstract away several details of networking hardware. Although this is a boon for the network programmer, the burden now shifts to the compiler and runtime system to provide a robust, high-level abstraction. Unsurprisingly, several compilers and runtime systems suffer subtle but serious bugs.

We developed a Coq-certified compiler and runtime system for NetCore (the intellectual predecessor of NetKAT) that is proven correct with respect to a detailed operational model of a network [9]. Specifically, we developed a weak bisimulation between a NetCore abstract machine and the network model. The abstract machine models a function that simply moves packets from one switch to another in big steps, but the network model details how packets are processed within network devices. For example, it models the I/O buffers on a switch, idiosyncrasies of packet-parsing, and several sources of non-determinism in π-calculus style. We extracted the controller to OCaml, benchmarked its performance, and found that it is competitive with other platforms.

Other Work I've worked on several other projects that either extend or leverage my primary research that is discussed above. FatTire is a network programming language that makes it easy to construct fault-tolerant forwarding fabrics [26]. Using FatTire, the programmer writes a regular expression that describes a set of possible paths and the compiler automatically calculates backup routes from that set to route around potential link failures. I was involved in early work on FlowLog, which is a network programming language that allows the programmer to write stateful packet-processing functions using a variant of Datalog [22]. Finally, PANE is a platform that allows applications running on end hosts to improve their own performance

by reconfiguring the network, within limits set by the network operator [6, 5]. In a series of experiments, we showed that PANE can dramatically improve the performance of canonical applications, such as Hadoop and ZooKeeper, with only minor changes to have them communicate their intent to the PANE controller.

Configuration Management

System administrators manage both networking and computing resources. Just as SDN started a revolution in network management, several domain-specific languages have had a dramatic effect on system configuration management over the past few years. Earlier, it may have been feasible to manage system configurations directly from the command line by running installers, editing configuration files, setting permissions, etc. But, the sheer scale of modern datacenters and cloud computing environments has made system configuration languages essential for computing infrastructure to be reliable and secure.

In recent work [3], I've investigated a system configuration language called Puppet, which is used by hundreds of organizations to manage thousands of machines [4]. Puppet 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 specifies a collection of resources (e.g., packages, user accounts, and files), their desired state (e.g., installed or not installed), and the dependencies between them. Although Puppet performs some static checking, there are many opportunities for errors to occur.

A key issue with Puppet is that configurations can easily be non-deterministic. Puppet configurations are often under-constrained (to allow them to be composed with each other) and underspecified (to allow them to be applied to a variety of machine states). However, if the user fails to specify all necessary constraints, the configuration becomes non-deterministic, which could be disastrous in practice. i.e., a configuration that works on a test machine may raise an error on a production machine or even silently mis-configure the production machine.

We developed a sound, complete, and scalable algorithm to verify that configurations are deterministic. To the best of our knowledge, our work is the first formal investigation of Puppet, so we began by developing a Puppet semantics that models resources as programs in a simple imperative language with filesystem operations. By taking this approach, the subtle interactions between different resource types become clear. Next, we encoded these programs as formulas in an SMT solver, applying partial order reduction and program slicing to keep the encoding small. Finally, we built an SMT-based determinism checker, applied it to several real-world Puppet configurations, and found determinacy bugs.

A Puppet configuration that is provably deterministic can be viewed as a function over the state of the system. This function can be analyzed and manipulated in several other ways. For example, when a machine is updated to a new configuration, it is easy for the machine state and its configuration to "drift" apart: an update from version 1 to version 2 may not have the same effect as applying version 2 to a new machine. To address this problem, we built a simple program synthesis tool that calculates an update to version 1 that has the same effect that version 2 would have had on the original machine.

Future Work

I plan to continue working on problems that arise in network programming languages and system configuration languages. These language technologies are still in their infancy, so there is a unique opportunity for programming languages research to positively affect their development. Moreover, several open research questions still remain. Future directions for my research include the following:

- Today's network programming languages can only be used in networks under the control of a single administrator (e.g., an enterprise or datacenter network). They cannot be used on larger inter-domain scales, where operators are stuck using low-level mechanisms for observing (e.g., traceroute) and controlling (e.g., selective advertising) routes. Can we build high-level abstractions for programming and monitoring inter-domain networks? Moreover, can we so in a manner that is secure and does not require a "fork-lift" upgrade to Internet infrastructure?

- Puppet and related languages allow machines to be configured using reusable, compositional abstractions, but there is a trade-off. In several situations, a one-off terminal command may be easier than modifying a Puppet configuration. However, to do so would break the abstractions the language provides. Can we have the best of both worlds and automatically keep a Puppet configuration in sync with low-level changes to the system? The challenge is to do so while preserving the abstractions that the user has developed.

- Network and system configuration are intimately related tasks, but they are typically tackled with a heterogeneous set of tools. This makes it difficult to reason about end-to-end properties that involve both the network configuration and the software stack. I believe that this is the right time to build a language for cloud computing that expresses network configurations, system configurations, and application logic. The key advantage of a linguistic approach is that it would provide a unified framework for reasoning about application performance and security.

References

[1] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic Foundations for Networks. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL),* 2014.

[2] Ravi Chugh, David Herman, and Ranjit Jhala. Dependent Types for JavaScript. *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA),* 2012.

[3] Joseph Collard, Nimish Gupta, Rian Shambaugh, Aaron Weiss, and Arjun Guha. On Static Verification of Puppet System Configurations. Draft available at `http://arxiv.org/abs/1509.05100`.

[4] Puppet Customers. `puppetlabs.com/about/customers`.

[5] Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Hierarchical Policies for Software Defined Networks. *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN),* 2012.

[6] Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Participatory Networking: An API for Application Control of SDNs. *ACM SIGCOMM Conference,* 2013.

[7] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A Network Programming Language. *ACM International Conference on Functional Programming (ICFP),* 2011.

[8] Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. Fully Abstract Compilation to JavaScript. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL),* 2013.

[9] Arjun Guha, Mark Reitblatt, and Nate Foster. Machine Verified Network Controllers. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI),* 2013.

[10] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The Essence of JavaScript. *European Conference on Object-Oriented Programming (ECOOP),* 2010.

[11] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing Local Control and State Using Flow Analysis. *European Symposium on Programming (ESOP),* 2011.

[12] Arpit Gupta, Laurent Vanbever, Muhammad Shahbaz, Sean Donovan, Brandon Schlinker, Nick Feamster, Jennifer Rexford, Scott Shenker, Russ Clark, and Ethan Katz-Bassett. SDX: A Software Defined Internet Exchange. *ACM SIGCOMM Conference,* 2014.

[13] Tim Hinrichs, Natasha Gude, Martin Casado, John Mitchell, and Scott Shenker. Practical Declarative Network Management. *ACM SIGCOMM Workshop on Research on Enterprise Networking (WREN),* 2009.

[14] U.S. Senate Subcommittee on Investigations. *Online Advertising and Hidden Hazards to Consumer Security and Data Privacy.* Available at `www.hsgac.senate.gov/download/?id=3B38A382-8E10-4527-904C-24F37A0D6220.` 2014.

[15] Seonghoon Kang and Sukyoung Ryu. Formal specification of a JavaScript module system. *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA),* 2012.

[16] Dexter Kozen. Kleene Algebra with Tests. *ACM Transactions on Programming Languages and Systems (TOPLAS),* 19(3):427–443, 1997.

[17] Benjamin S. Lerner, Liam Elberty, Jincheng Li, and Shriram Krishnamurthi. Combining Form and Function: Static Types for JQuery Programs. *European Conference on Object-Oriented Programming (ECOOP),* 2013.

[18] Magnus Madsen, Frank Tip, and Ondrej Lhoták. Static Analysis of Event-Driven Node.js JavaScript Applications. *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA),* 2015.

[19] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A Compiler and Run-time System for Network Programming Languages. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL),* 2012.

[20] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing Software-Defined Networks. *USENIX Symposium on Networked System Design and Implementation (NSDI),* 2013.

[21] Tim Nelson, Andrew D. Ferguson, Michael J. G. Scheer, and Shriram Krishnamurthi. Tierless Programming and Reasoning for Software-Defined Networks. *USENIX Symposium on Networked System Design and Implementation (NSDI),* 2014.

[22] Tim Nelson, Arjun Guha, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. A Balance of Power: Expressive, Analyzable Controller Programming. *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN),* 2013.

[23] Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. ADsafety: Type-based Verification of JavaScript Sandboxing. *USENIX Security Symposium,* 2011.

[24] Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. Semantics and Types for Objects with First-Class Member Names. *Workshop on Foundations of Object-Oriented Languages (FOOL),* 2012.

[25] Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. Type-based Verification of JavaScript Sandboxing. *Journal of Computer Security,* 22(4):511–565, 2014.

[26] Mark Reitblatt, Marco Canini, Arjun Guha, and Nate Foster. FatTire: Declarative Fault Tolerance for Software Defined Networks. *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN),* 2013.

[27] Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. Concrete Types for TypeScript. *European Conference on Object-Oriented Programming (ECOOP),* 2015.

[28] Steffen Smolka, Spiridon Aristides Eliopoulos, Nate Foster, and Arjun Guha. A Fast Compiler for NetKAT. *ACM International Conference on Functional Programming (ICFP),* 2015.

[29] Andreas Voellmy and Paul Hudak. Nettle: Functional Reactive Programming of OpenFlow Networks. *padl,* 2011.

[30] Andreas Voellmy, Junchang Wang, Y. Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying SDN Programming Using Algorithmic Policies. *ACM SIGCOMM Conference,* 2013.