RESEARCH STATEMENT

JEAN YANG

The goal of my research is to help programmers build more reliable software with less effort. Towards this goal, I developed a programming model that allows selective automation of specific concerns across the program. This model integrates high-level rules into a familiar paradigm to reduce programmer burden without requiring a new way of reasoning about the entire program. My research is about designing language constructs and execution strategies that allow the runtime to take responsibility for tedious, error-prone tasks that cut across the program.

I have demonstrated how to use this programming model to address the problem of data leaks and privacy breaches. As more information becomes digital, programmers not only spend more time enforcing policies for privacy and security, but also make more mistakes. To solve the problem of missing security checks across the program, I created **policy-agnostic programming, a paradigm that separates the implementation of information flow policies from the rest of the program and automates their enforcement**. I have implemented this paradigm in a programming language with strong theoretical guarantees that I have used to build web applications, including a conference management system that has run an ACM-sponsored workshop.

1. Experience in Verifying Software Correctness

I began thinking about the power of language-level guarantees when I worked with Chris Hawblitzel at Microsoft Research to verify the Verve operating system [12, 13]. In Verve, we leverage language guarantees to verify end-to-end type and memory safety properties. Traditionally, operating systems are written in low-level languages with few guarantees. This is because they have behaviors–such as context switching–that cannot be implemented in type-safe languages. In Verve, we isolated the subset of the operating system kernel that is necessarily low-level and verified it with respect to a specification of end-to-end type and memory safety. The specification allows us to implement the rest of the operating system in a type-safe language, thus simplifying the verification task. Our 2010 paper [12] won Best Paper Award at PLDI and was republished as an invited Research Highlight in the Communications of the ACM [13].

After Verve, I began to think about how to design languages to automate concerns other than memory management. I was especially interested in information flow, a global and error-prone concern. I contacted Nikhil Swamy and Juan Chen at Microsoft Research to work on type-based verification in Fine. The Fine language verifies security policies using dependent types (types that depend on program values). Fine has a proof-carrying compilation process that automatically generates proofs that can be stored with bytecode. I contributed to theory and practice to enable systems using Fine to pass around proof-annotated bytecode in a distributed setting [10, 11].

While working on Fine, I confirmed that verification is not enough to help programmers write secure software. In existing languages, the programmer is responsible for enforcing information flow policies explicitly wherever sensitive data is used. For example, to enforce the policy only my friends who are nearby can see my GPS location, the programmer must enforce the policy wherever the users location value flows, including through any computations. A mistake in implementing any of these checks can result in an information leak. While languages like Fine reject programs that leak information, it remains difficult to write these programs in the first place. Enforcing policies automatically would reduce programmer burden.

2. A Language for Automatically Enforcing Information Flow Policies

During my Ph.D., I created Jeeves [14, 3], a programming language that automatically enforces information flow policies. Jeevess policy-agnostic programming model asks the programmer to associate information policies directly with sensitive data while relying on the runtime system to enforce the policies. In Jeeves, the programmer defines different views for sensitive values, along with policies about when and to whom the values may be shown. For each sensitive value, the runtime simulates simultaneous executions to show the appropriate result based on the viewer. For the GPS location example, the programmer might create versions of the location corresponding to the precise GPS location and the current country, associate a policy that says only nearby viewers may see the value. The rest of the program may be policy-agnostic: code that finds and shows all the users in a given location does not need to be aware of the policies. The resulting programs are correct by construction with respect to the stated policies. With Jeeves, the programmer no longer needs to implement access checks across the program.

I have characterized a theoretical foundation for Jeeves and built a usable implementation. Jeeves has a big-step dynamic semantics that guarantees 1) private values do not leak to public viewers (non-interference) and 2) the system correctly determines the access level of a viewer (policy compliance). The semantics takes into account state updates and implicit flows (potential leaks through control flow involving sensitive values). In order to support database-backed applications, I extended Jeeves to create Jinq, a language with with policy-agnostic, integrated queries. I have used Jinq to build a web framework, Jacqueline, which provides end-to-end guarantees across the frontend, application, and database. I have implemented Jeeves in both Scala and Python and Jacqueline as an extension of the Django web framework. I have used Jacqueline to implement a conference management system used to run an ACM-sponsored workshop (PLOOC 2014).

The ideas in Jeeves have impact beyond my research collaborators. Our POPL [14] and PLAS [3] papers about the design and semantics of Jeeves have been part of the curriculum for a graduate-level language-based security course at the University of Maryland. Researcher Eva Rose has developed a Haskell implementation of Jeeves [9]. In the last year, over 100 developers have bookmarked the repository for the Python embedding of Jeeves on the web-based repository hosting service GitHub. In addition to impact within the academic and programming communities, we have had popular press coverage in *New Scientist* [2], *Gigaom* [5, 6], *MIT Technology Review* [4], *Wired* [7], *Fast Company* [8], and *Fast CoLabs* [1].

3. Future Directions

In addition to expanding the expressiveness and usability of the policy-agnostic approach for information flow policies, I am interested in generalizing the approach to other concerns, in particular domain-specific concerns in social science and biology.

3.1. Providing system-wide security guarantees.

Current implementations of policy-agnostic programming assume that both the runtime and the database are trusted. In order to provide realistic security guarantees, we need to reduce the amount of trust. I plan to do this by efficiently integrating cryptography. This involves carefully choosing the kinds of encryption to use, as well as the parts of the system to encrypt, for instance the database. By factoring out the enforcement of information flow policies, the policy-agnostic model presents unique challenges and opportunities for removing trust. There may be, for instance, connections between faceted execution and functional encryption, a form of public-key encryption that allows a secret key holder to learn a function of what is encrypted without learning anything else about the program. I have been discussing these ideas with cryptographers, including Vinod Vaikuntanathan at MIT.

3.2. Expanding policies to handle aggregate values.

Information flow is often too strong a requirement for the properties that programmers want. For instance, information flow policies cannot permit an average salary to be revealed while protecting individual salaries. To support such policies, I plan to expand policy-agnostic programming to enforce policies based on aggregate values. One way to do this is to incorporate differential privacy, which allows an aggregate value to be revealed if the likelihood of inferring individual values is low. Another potential solution is to support policies on computation histories, for instance allowing values to be revealed if certain operators have been applied to derive the result. Runtime support for this would be a natural extension of Jeevess dynamic multi-execution approach. I have discussed these ideas with Nickolai Zeldovich and Raluca Popa at MIT, as well as with Fred Schneider at Cornell, who is developing a theoretical framework for policies over computational histories.

3.3. **Making policy-agnostic programming usable by non-computer scientists.** Journalists are increasingly interested in uncovering patterns and misdeeds through analyzing data. For some of this data analysis, for instance in the case of finding medical errors, it is critical that sensitive information from individuals does not get leaked. If trained developers have trouble correctly implementing privacy policies, we cannot expect journalists to get it right. To help journalists query semi-private data, I want to make policy-agnostic programming accessible for people who are not trained as computer scientists. In addition to creating a scalable execution model that works with realistic data set sizes, I plan to create tools, for instance a policy verifier and a policy visualizer, that make it easier to create and reason about policies and programs. I became interested in this goal after taking a course on the impact of technology on journalism. I have continued working with journalists in NeuWrite Boston, a working group that I run comprised of scientists and science writers. Members include editors from *MIT Technology Review* and *NOVA*.

3.4. **Separating program concerns to model biological systems.** Researchers have demonstrated advantages in modeling biological systems operationally (using logical programs) rather than denotationally (using equations). The logical models of biological systems often suffer from poor performance. Additionally, implementing logical models often requires biologists to learn an unfamiliar programming model. Using a policy-agnostic paradigm to model these systems would allow the programmer to describe part of the system logically and everything else more procedurally. Because the programmer specifies the control flow, this potentially mitigates some performance issues. In addition, the models could look more similar to programs biologists are accustomed to writing. Something like the Python implementation of Jeeves, for instance, could integrate with existing molecular dynamics modeling packages in Python. I have been talking with Benjamin Hall at the University of Cambridge about the feasibility of using policy-agnostic programming to model protein signaling systems, where policies describe communication between different parts of a cell.

3.5. **Concluding thoughts.** By demonstrating that programming languages can take responsibility for global concerns, I hope to change the way language designers think about what is possible to automate in programs. The policy-agnostic approach is a step towards automating what is repetitive and error-prone in programming.

References

[1] Tina Amirtha. Non-techie ways to prevent your company from suffering the next Heartbleed bug. *Fast CoLabs*, May 2014.

[2] Jacob Aron. What your online friends reveal about where you are. *New Scientist*, January 2012.

[3] Thomas H. Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. Faceted execution of policy-agnostic programs. 2013.

[4] CSAIL. New programming language removes human error from privacy equation. *MIT Technology Review*, February 2014.

[5] Barb Darrow. Mission possible? Jean Yang. *Gigaom*, May 2013.

[6] Barb Darrow. Want to build privacy into your apps? Check out Jeeves, now available in Python. *Gigaom*, February 2014.

[7] Klint Finley. Out in the open: A new programming language with built-in privacy protocols. *Wired*, March 2014.

[8] Jessica Leber. A better way to protect privacy? Take the programmer out of the equation. *Fast CoExist*, March 2014.

[9] Eva Rose. Constraint generation for the Jeeves privacy language. Technical Report MIT-CSAIL-TR-2014-020, Massachusetts Institute of Technology, October 2014.

[10] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. 2011.

[11] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. *Journal of Functional Programming*, 23, 7 2013.

[12] Jean Yang and Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. 2010.

[13] Jean Yang and Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. *Commun. ACM*, 54(12), 2011.

[14] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. 2012.