Research Statement

Fan Long

November 2016

My areas of interest are software engineering, programming languages, and systems security. My research to date has focused on developing automated programming systems that combine program analysis, runtime systems, and machine learning to improve software reliability and security. I have developed systems that automatically identify and eliminate errors in large software programs and systems that enable software programs to operate successfully in spite of the presence of errors. The systems I developed include the first patch generation system that learns from successful human patches to automatically fix software defects [2], the first sound input filtering system that nullifies critical integer overflow errors [10], and the first input rectification system that automatically rectifies potentially malicious inputs [12]. These systems significantly advance the state of the art and/or enable completely new approaches for improving software reliability and security.

Motivation and Overview

Software applications are integrated into every part of our society. There are more software programs than ever before, e.g., the number of open source repositories in GitHub reached 12 million at 2015, and this number continues to grow. As the dependence of our society on these programs also continues to grow, making these programs reliable and secure becomes an increasingly important challenge for our society and a daunting task for software developers. Automated techniques for improving software reliability and security will be even more important than ever before.

My research goal is to develop automated techniques to improve the reliability and security of real world software applications. My research realizes this goal from three angles:

- **Identify and Fix Code Defects:** One way to improve the reliability and security of a program is to eliminate defects in its source code. I have worked on both bug finding systems that automatically identify defects [8, 14] and patch generation systems that automatically generate correct patches to fix defects [1, 2, 5–7, 15]. Most notably, I developed Prophet, a novel patch generation system that learns from past successful human patches to guide the patch generation process [2]. The experimental results show that Prophet significantly outperforms all previous techniques and advances the state of the art for patch generation [2].

- **Enable Recovery from Fatal Errors:** Another way to improve reliability and security is to enable a program to recover from runtime errors during its execution. I developed RCV [9], a lightweight program recovery system. The experimental results show that RCV effectively enables off-the-shelf binary programs to recover from otherwise fatal errors with negligible overhead.

- **Identify and Rectify Malicious Inputs:** Yet another way to improve reliability and security is to identify potentially malicious inputs that can exploit vulnerabilities in a program. We can then either discard or rectify (i.e. change the inputs so that the changed inputs cannot exploit the vulnerabilities) the identified malicious inputs. I have developed several input filtering and rectification techniques to explore both options [4, 10, 12]. The experimental results show that my techniques, in many cases with guarantees, nullify target vulnerabilities in software programs while having few or no false positives in practice [4, 10].

I have also worked on other aspects of security and software engineering, including security attacks [3], record and replay tools [13], and automatic code generation from natural language descriptions [11]. My research has received wide media coverage and spearheaded two successful DARPA programs, Mission-oriented Resilient Clouds (MRC) and Mining and Understanding Software Enclaves (MUSE).

Automatic Patch Generation

Automatic patch generation holds out the promise of automatically correcting software defects without the need for human developers to diagnose, understand, and correct these defects. The prominent generate-and-validate approach starts with a test suite of inputs, at least one of which exposes a defect in the software. It then generates a space of candidate patches and searches this space to find validated patches that produce correct outputs for all inputs in the test suite.

One important challenge is the weak test suite issue. Because the supplied test suite typically does not exercise the full functionality of the program, previous patch generation systems often generate validated but incorrect patches that produce incorrect outputs for inputs that are not covered by the test suite [6]. Another important challenge is the search space trade-off between coverage and the tractability [1]. On one hand, the search space needs to be large enough to contain correct patches for the target class of errors. On the other hand, the search space needs to be small enough so that the patch generation system can efficiently explore the space to find the correct patches [1].

Prophet. To overcome the weak test suite issue, I developed Prophet [2], a novel generate-and-validate patch generation system that learns from a training set of successful human patches obtained from open source software repositories. Prophet first learns a probabilistic application-independent model of correct code from the training patches. The learned model assigns a probability score to each candidate patch in the Prophet search space. Prophet then uses the model to rank candidate patches in order of likely correctness, prioritizing potentially correct patches among many validated patches.

A key challenge for Prophet is to identify and learn universal properties of correct patches. Prophet extracts not only the characteristics of the patch itself, but also interactions between the patch and surrounding code. Prophet appropriately abstracts away syntactic details such as variable names to produce a set of application-independent features. Experimental results show that, on a benchmark set of 69 real-world defects systematically collected from eight large open-source projects, Prophet significantly outperforms previous systems and generates correct patches for up to nine times more defects.

Genesis. To navigate the challenging search space trace-off, I am working on a new patch generation system called Genesis [15]. Unlike Prophet, which uses a set of manually defined code transform rules to derive its search space, Genesis automatically infers useful code transforms from a large database of human patches. The inferred transforms summarize common patterns of the patches that developers applied to fix defects. Genesis then formulates the search space trade-off between coverage and tractability as an integer linear programming problem and uses an off-the-shelf solver to select a set of the inferred transforms into the final search space for patch generation. Combined with the previous patch ranking techniques in Prophet, the ultimate goal of my research is to build a fully automated and data-driven patch generation system that combines learned human expert knowledge with machine computational power.

CodePhage. I also explored techniques that directly transfer useful code across applications for the scenarios where correct code in one application is able to eliminate errors in another application. CodePhage [7] automatically locates correct code in a donor application and then transfers that code as a patch to eliminate defects in a recipient application. CodePhage first uses a symbolic execution technique to extract the relevant program logic of the correct code into an application-independent symbolic expression tree. CodePhage then uses a solver-based synthesis technique to convert the extracted expression tree into a new patch for the recipient application. CodePhage has been applied to transfer code across seven large real world applications, eliminating ten fatal integer overflow, buffer overflow, and divide by zero errors.

Contributions. Prophet [2] is the first patch generation system that learns from past successful patches. It implements a novel framework that combines statistical inference and program analysis to learn a probabilistic model of correct patches. Genesis [15] provides the first inference algorithm that infers code transforms and patch generation search spaces from successful patches. CodePhage [7] is the first system that transfers code across applications.

Program Recovery

When a fatal runtime error occurs, the execution of a software program terminates immediately. Obvious potential negative consequences include denial of service to users and data corruption. Given these consequences, a technique that enables the program to successfully survive the error would be of enormous value.

RCV. RCV [9] is a runtime recovery system that enables software programs to survive divide-by-zero and null-dereference errors. RCV operates directly on off-the-shelf stripped x86 binary executables. It replaces the standard divide-by-zero (SIGFPE) and segmentation violation (SIGSEGV) signal handlers with its own handlers. When an error occurs, the RCV signal handler skips the error triggering operation and manufactures the default value of zero as the result of the operation if necessary.

One concern of previous recovery techniques is that the recovery may cause undefined behaviors and collateral damage to the whole software system. To address this concern, RCV implements recovery shepherding, an influence tracking and error containment system. The system tracks all values influenced by the manufactured values that RCV generates. It also detects when all influenced state has either been deallocated or overwritten with clean values. Except during recovery, RCV does not modify the behavior of the program at all and therefore incurs negligible overhead. The experimental results show that for 17 of the total 18 systematically collected errors from seven real world applications, RCV enables the application to continue to execute to provide acceptable output and service to its users on the error-triggering inputs. For 13 of the 18 errors, the continued RCV execution eventually flushes all of the repair effects and RCV detaches to restore the application to full functionality.

Contributions. RCV is the first lightweight recovery system that enables off-the-shelf binary applications to recover from otherwise fatal null-dereference and divide-by-zero errors [9].

Input Filtering and Rectification

Deploying input filters is a standard approach to protect a vulnerable program. Input filters use a set of input constraints to identify and discard potentially malicious inputs that can exploit vulnerabilities in a program. Such constraints are typically either specified manually or inferred from previous inputs. One issue with previous approaches is that the constraints do not come with a guarantee to identify all malicious inputs. This absence of the guarantee leaves the possibility of exploitation. Another issue is that the constraints often incorrectly identify atypical but benign inputs as malicious inputs, causing potential loss of desirable data.

SIFT. To address the first issue, I developed SIFT [10], a sound input filter system that guarantees to nullify integer overflow errors. SIFT starts with a set of critical expressions from memory allocation and block copy sites. These expressions control the sizes of allocated or copied memory blocks at these sites. The core of SIFT is an interprocedural, demand-driven, precondition static analysis that propagates the critical expression backwards against the control flow. Unlike previous techniques that unroll loops causing unsound results, the analysis uses novel abstraction and normalization algorithms that exploit the computation pattern of critical expressions to automatically obtain loop invariants. SIFT uses the analysis to obtain a symbolic condition that captures all expressions that the program may evaluate (in any execution) to obtain the values of critical expressions. SIFT then generates filters to check new inputs against the symbolic condition. The experimental results show that SIFT successfully generates sound input filters, within one second, for 56 out of 58 memory allocation and block memory copy sites in five real-world applications. These sound filters nullify six known vulnerabilities with no false positives in practice [10].

SOAP. To address the second issue (i.e., the potential loss of desirable data), I developed SOAP [12], an automatic input rectification system. Unlike standard input filtering techniques, which discard atypical inputs and cause data loss, SOAP rectifies (i.e., changes) the atypical inputs and passes rectified inputs to the program for presentation to the user. It may therefore deliver much or even all of the desirable data present in those atypical but benign inputs.

SOAP first uses dynamic taint analysis to identify those input fields that are related to critical operations during the execution of the program (i.e., memory allocations and memory writes). SOAP infers two kinds of constraints over the identified critical input fields, bound constraints over the values or the sizes of the fields and correlated constraints that capture relationships between the values of integer fields and the lengths of (potentially nested) string or raw data fields. SOAP then enforces the inferred constraints on an incoming input with an iterative rectification algorithm. The experimental results show that SOAP can effectively nullify all six evaluated vulnerabilities while preserving desirable data in the inputs. On five evaluated media programs, less than 1.6% of inputs are rectified and over 75% of the rectified images or videos appear unchanged to humans [12].

Contributions. SIFT [10] is the first sound input filtering technique for integer overflow errors. It provides a novel interprocedural precondition analysis to soundly analyze critical integer expressions in software programs. SOAP [12] is the first automatic input rectification system.

Future Research

Exploit the Growing Volume of Programs. My research already demonstrates that the growing volume of programs is not just a challenge but a great opportunity. This opportunity enables exciting new learning, inference, and code transfer techniques that were impossible before [2, 7, 15]. I plan to continue my research on such new techniques that can exploit the growing volume of existing software programs. The intuition is that previous development efforts for solving software engineering problems in existing programs can provide valuable human knowledge. Exploiting such human knowledge can enable new automated techniques to solve the problems in a new program, especially if the new program implements similar functionality as some of the existing programs. The ultimate goal of this research direction is to leverage the combined expertise and past efforts of developers worldwide to better solve future software engineering problems.

New techniques in this research direction often require acquiring training databases from existing programs. Most of the new techniques will require automated compilation and testing of a training program. Some techniques, like Prophet [2], will also require collecting previous patches of a training program. I plan to develop new software engineering solutions to collect and maintain such training databases. The repository systems of existing programs are often unsatisfactory because critical information for successfully collecting the programs (e.g., how to compile the programs and which revision fixes a defect) is only available in natural language as documentation and commit logs. My past research provides a novel framework for automatically generating input parser programs from natural language descriptions [11]. I plan to extend the framework to generate repository processing scripts from documentation to help collect training databases.

Bring New Functionality from Existing Code For almost any desirable functionality, developers now can find existing code snippets from Q/A websites such as stack overflow, from documentation of third party libraries, or from other open source projects. Instead of developing the desired functionality from scratch, a more common practice is to copy an existing code snippet and appropriately integrate it with the host code under development. This manual development process includes wiring up correct data flow paths between the copied code and the host code. It also includes inserting necessary glue code at the beginning and end of the copied code to initialize and clean up relevant data objects. A technique that automates even parts of this development process will be extremely valuable.

I plan to investigate new techniques that help developers synthesize the implementation of new functionality from existing code. One potential technique is to first use program analysis to convert an existing code snippet and a host code snippet into two application-independent automata graphs. The automata graphs capture data flow paths and internal states of relevant objects that are critical for the success of the integration. The graphs also abstract away program-specific syntactic details such like variable names. Integrating the example code into the host code can be formulated as a prediction problem of finding the most promising way to connect two graphs. It is then possible to use standard machine learning algorithms to solve this prediction problem.

Develop Collective Security Techniques. Large software programs often contain components that implement similar functionality and therefore these programs may share similar vulnerabilities. I plan to develop new collective techniques that exploit this similarity to improve software security. Specifically, I plan to develop collective vulnerability detection techniques. If a zero-day vulnerability is exploited in one program, such a technique uses the exploitation information to synthesize new inputs to detect (i.e., exploit) similar vulnerabilities in other programs. I also plan to continue my research on patch transfer techniques like CodePhage [7]. If developers write a patch to fix a zero-day vulnerability, such a technique automatically transfers the patch to other applicable programs to fix similar vulnerabilities. The goal is to enable software programs to form a security alliance in which security alerts, updates, and development efforts are automatically shared among all applicable programs.

Unify Formal Reasoning and Machine Learning. Many existing approaches directly apply successful learning techniques from other domains to software programs (e.g., applying techniques from natural language processing as if the program is a series of word tokens). Such approaches often produce unsatisfactory results, because they often cannot capture semantic level information of the programs and therefore only learn shallow syntactic level knowledge. I believe unified approaches that combine the power of formal reasoning and machine learning are required for a breakthrough.

I plan to design new analysis approaches that unify traditional program analysis techniques based on formal reasoning and emerging new techniques based on machine learning. For example, for analyzing software programs, the formal reasoning techniques first extract semantic information from each of the programs and abstract such information appropriately into a model amenable to learning techniques. The learning techniques then process the abstracted models of all programs in a training database together to learn useful statistical knowledge. I have successfully applied such approaches on specific software engineering problems like patch generation [2] and parser generation [11]. An intriguing open question I want to investigate is then how to build an universal framework for general software analysis and prediction tasks.

Referenced Publications (See CV for Complete List)

[1] Fan Long and Martin Rinard. "An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems". In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE 2016. Austin, Texas: ACM, 2016, pp. 702–713.

[2] Fan Long and Martin Rinard. "Automatic Patch Generation by Learning Correct Code". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL 2016. St. Petersburg, FL, USA: ACM, 2016, pp. 298–312.

[3] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. "Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity". In: *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. CCS 2015. Denver, Colorado, USA: ACM, 2015, pp. 901–913.

[4] Brendan Juba, Christopher Musco, Fan Long, Stelios Sidiroglou-Douskos, and Martin C. Rinard. "Principled Sampling for Anomaly Detection". In: *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. 2015.

[5] Fan Long and Martin Rinard. "Staged Program Repair with Condition Synthesis". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: ACM, 2015, pp. 166–178.

[6] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. "An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems". In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ISSTA 2015. Baltimore, MD, USA: ACM, 2015, pp. 24–36.

[7] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. "Automatic Error Elimination by Horizontal Code Transfer Across Multiple Applications". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2015. Portland, OR, USA: ACM, 2015, pp. 43–54.

[8] Stelios Sidiroglou-Douskos, Eric Lahtinen, Nathan Rittenhouse, Paolo Piselli, Fan Long, Deokhwan Kim, and Martin Rinard. "Targeted Automatic Integer Overflow Discovery Using Goal-Directed Conditional Branch Enforcement". In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS 2015. Istanbul, Turkey: ACM, 2015, pp. 473–486.

[9] Fan Long, Stelios Sidiroglou-Douskos, and Martin Rinard. "Automatic Runtime Error Repair and Containment via Recovery Shepherding". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2014. Edinburgh, United Kingdom: ACM, 2014, pp. 227–238.

[10] Fan Long, Stelios Sidiroglou-Douskos, Deokhwan Kim, and Martin Rinard. "Sound Input Filter Generation for Integer Overflow Errors". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL 2014. San Diego, California, USA: ACM, 2014, pp. 439–452.

[11] Tao Lei, Fan Long, Regina Barzilay, and Martin C. Rinard. "From Natural Language Specifications to Program Input Parsers". In: *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics, ACL 2013, 4-9 August 2013, Sofia, Bulgaria, Volume 1: Long Papers*. 2013, pp. 1294–1303.

[12] Fan Long, Vijay Ganesh, Michael Carbin, Stelios Sidiroglou, and Martin Rinard. "Automatic Input Rectification". In: *Proceedings of the 2012 International Conference on Software Engineering*. ICSE 2012. Zurich, Switzerland: IEEE Press, 2012, pp. 80–90.

[13] Ming Wu, Fan Long, Xi Wang, Zhilei Xu, Haoxiang Lin, Xuezheng Liu, Zhenyu Guo, Huayang Guo, Lidong Zhou, and Zheng Zhang. "Language-based Replay via Data Flow Cut". In: *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2010. Santa Fe, New Mexico, USA: ACM, 2010, pp. 197–206.

[14] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. "MODIST: Transparent Model Checking of Unmodified Distributed Systems". In: *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*. NSDI 2009. Boston, Massachusetts: USENIX Association, 2009, pp. 213–228.

Technical Report

[15] Fan Long, Peter Amidon, and Martin Rinard. *Automatic Inference of Code Transforms and Search Spaces for Automatic Patch Generation Systems*. Tech. rep. MIT-CSAIL-TR-2016-010. 2016.