Research Statement

Gary Wassermann
wassermg@cs.ucdavis.edu

I seek to advance scientific knowledge on security and reliability of software, primarily by using PL techniques to address important problems in security and software engineering. In working toward this goal, I developed static analyses for web application vulnerabilities, defined and introduced work on a new class of a decision problems, and contributed to a virtual machine-based analysis of time-dependant malware. As my past experience reflects, once I identify an important real-world problem, I formalize it and develop and implement an algorithm to solve it; this process involves exploration in areas ranging from formal logic to system implementation.

My current research goal is to make web programming as safe and reliable as desktop programming in Java. The challenge is that web applications are dynamic and distributed, so at no point does all the code appear for any one entity to regulate. However, safety is essential because anyone with a web browser can access applications that handle many people's sensitive information.

Web applications differ from desktop programs in fundamental ways, including their architecture, languages, common platforms, and patterns of interaction, and these differences add opportunities for errors. Indeed, even though web application languages provide memory safety and thus prevent historically significant errors such as buffer overflows, web applications face classes of errors that top the charts for both prevalence and impact. My past work addresses problems that arise from their architecture; that is, from their being constructed in multiple tiers that communicate via structure-less strings. My future work will address problems arising from the other differences named above.

Current Research

The topic I primarily focussed on for my dissertation research is input validation, in which errors can allow both cross-site scripting (XSS) and SQL injection. For the past several years, XSS and SQL injection have been the two most frequently reported classes of security vulnerabilities, and single instances have aversely affected hundreds of thousands of users. Checking input validation requires analyzing not only a web application's use of filters and transformations to make arbitrary string values "safe" in a constructed command, but also the way in which the constructed command will be interpreted.

Characterizing Command Injection. Initially, most material presented SQL injection by example and described ad hoc solutions. Along with Zhendong Su, I proposed the first principled definition of SQL injection by describing the effect that untrusted input is permitted to have on SQL queries that a web application constructs [*POPL 2006*]. Our characterization employs two primary notions: sentential forms, an abstraction from parsing algorithms used in compilers; and integrity, a fundamental concept in security that forbids untrusted data from modifying trusted data. Our definition provides a solid foundation for designing web database APIs as well as checking for attacks and vulnerabilities.

Checking at Runtime and at Compile Time. Our sentential forms-based definition suggests a parsing-based runtime enforcement mechanism, but parsing alone does not suffice because injection attacks must parse under the SQL grammar to succeed. We needed the insight that delimiting untrusted (i.e., input) strings and adding the delimiters to the SQL grammar provides an elegant, sound, and complete way to distinguish safe queries from attacks [*POPL 2006*]. The principal runtime overhead of our technique is the cost of a single parsing, and, in our experiments, our runtime checks prevented all attacks and permitted all instances of normal usage.

Although our runtime checks prevent attacks, we also pursued static analysis because, in software development, static analysis reveals errors that may indicate broader problems, informs programmers about repeatable mistakes, and reduces runtime overhead by removing redundant checks. Previously, taint analysis served to find web application vulnerabilities, but taint analysis cannot guarantee the absence of vulnerabilities given our definition; it does not model string values or string functions' semantics. We modeled both by leveraging synergistically string analysis and taint-tracking [*PLDI 2007*]. Our analysis relies on novel adaptations of certain formal language algorithms, such as the classical algorithm for context-free language reachability, to track integrity levels and achieve soundness. Where a vulnerability exists, our analysis provides a witness in the form of an attack query. In our experiments, we successfully analyzed large (~160 KLoC) real-world programs, finding known and previously unknown vulnerabilities with a low false positive rate.

Finding Cross-Site Scripting Vulnerabilities. XSS is a related but fundamentally more difficult problem, in which an attacker exploits the trust a web client (browser) has for a trusted server and executes injected script on the browser with the server's privileges. Whereas database systems restrict command execution to a well-defined language, web browsers do not. Web browsers parse HTML permissively as a result of early software engineering decisions that seemed beneficial in the short term—they enabled browsers to display poorly written HTML pages—but have now made XSS more difficult to prevent. We examined browser source code and discovered many subtle and undocumented ways for untrusted strings to invoke the JavaScript interpreter. We then constructed a policy that describes these ways using a regular language and employed our string-taint analysis to find XSS vulnerabilities [*ICSE 2008*]. Our results are sobering: every manually written input validation routine that we analyzed and that allows any HTML fails to prevent XSS.

Testing Dynamic Applications. In the process of designing and implementing our static analysis, we found that many web applications use dynamic language features, and such features inhibit static analysis. We looked to testing as a complement to static analysis, but previous work on automated test input generation focusses on numeric values and pointer-based data structures, which languages like C and Java emphasize. Web scripting languages, like PHP, promote a style of programming that emphasizes strings and associative arrays. Along with Dachuan Yu and others at DoCoMo USA Labs, I designed and implemented an algorithm for automated test input generation for web applications [*ISSTA 2008*]. By incorporating values gathered from program executions, we model string operation semantics more precisely than static analyses have done, and we found vulnerabilities in real-world code that every available static analyzer failed to analyze.

Future Work

As previously mentioned, web applications differ from desktop applications not only in their architecture, resulting in the need for input validation, but also in their (1) patterns of close interaction—one server manages many user accounts and one browser manages accounts on many servers; (2) typical platforms—handheld devices increasingly support web access; and (3) languages—desktop-like applications and even desktop applications themselves are being written in web scripting languages. I intend to address the problems that these bring, while continuing to explore new problem domains for interdisciplinary work.

Client-side Information Integration. Web application mashups integrate data from multiple servers on one client, and if future web applications follow the current trajectory, the web browser will have to manage mutually distrustful code, much like an operating system does, yet with less control of the whole system and with more communication across boundaries. Whether data should be allowed to pass from one domain to another may depend on whether that data is executable, whether it is confidential, or how it will be used. I plan to investigate using a dynamic parameterized type system for JavaScript, where types are parameterized with domain identifiers, as a means to regulate communication while still allowing desired behavior.

Server-side Information Flow. Certain classes of web applications, particularly social networking sites, host many users' data on the same (logical) server. Often these sites cannot provide a separate domain to each user, so JavaScript's same-origin policy fails to prevent one user's JavaScript from accessing another user's data, thus enabling JavaScript worms. Such sites would normally like to enable users to enrich their pages with script and even share public views of JavaScript-based "desktops" with other users. By viewing data from two different users as having incomparable integrity labels, I want to design a mechanism that will both allow active content provided by users and prevent unwanted modifications that lead to such problems as JavaScript worms and JavaScript hijacking.

Concurrency. Handheld devices increasingly support web applications and they may soon surpass laptop computers as the preferred platform for web usage. Handheld devices' power limitations indicate that future generations of them will have many low-power processors, so for software to perform well on them, it will have to be parallelizable. I want to explore how to make client-side script parallelizable without introducing the errors so prevalent in classical concurrent programs. This setting differs from the standard setting in that typical client-side script includes a dynamic/meta-programming aspect, which a successful solution will still permit while adding appropriate restrictions.

Common JavaScript Programming Errors. As the line between the web and the desktop blurs, programmers increasingly use JavaScript for important programming tasks—already much of Firefox is written in JavaScript. This makes correctness important, but programmers tend to make different kinds of errors in a dynamic, prototype-based language than in C or Java. I plan to study and characterize common classes of errors, such as incorrect assumptions about the environment and unintended information flows, in order to design analyses and tools to prevent them.

My goal in this work is to help make the web support the functionality that today's desktops do while leveraging the unique benefits that the web provides. The work I do in the process may lead to a new language, since JavaScript, which was first designed simply to enhance user interfaces, is not the best language for its current task.

Interdisciplinary Work. I have also pursued research in several other branches of Computer Science. In *Theory*, I introduced a new class of decision problems, validity for languages of constraints, and, using novel constructions based on network flows and Helly's Theorem from computational geometry, gave a decision procedure for a setting that was inspired by my work on web applications [*FSTTCS 2006*]. In *Systems* and *Architecture*, I worked with Jed Crandall, Fred Chong, and others on VM-based analysis of evasive, time-dependent malware [*ASPLOS 2006*]. For that work, I designed a weakest precondition-based algorithm for discovering time-window boundaries of machine code behavior. I then helped to extend that work for a *Network Security* setting [*NOMS 2008*]. Related to *Databases*, I designed a type system for an XML query language, where the type system includes variable cardinality constraints. I look forward to collaborating with colleagues in other areas of computer science and with researchers in pure and applied sciences, to understand the requirements of domains other than the ones I am familiar with, and to investigate the possible use of analysis techniques for these domains.

Most broadly, my future research will address a range of issues in software engineering and software security and reliability. I want to investigate how to find bugs statically in large code bases, how to aid programmers in understanding software systems, and how to utilize and enforce implicit abstractions. This field continues to grow and develop rapidly, and I am excited to be on its forefront addressing the research challenges that it brings.