Research Statement

Ankur Taly

I am broadly interested in the applications of programming language theory, logic and formal methods to various real-world problems pertaining to web security, system security and verification. My research employs a two-pronged approach: (i) mathematically modeling real-world systems in order to rigorously formulate properties of interest in them, and (ii) developing analysis techniques for proving and disproving these properties. For instance, in [7], in order to verify that a JavaScript API confines security-critical resources, I rigorously formulated the confinement problem as a points-to analysis problem and then solved it by adapting off-the-shelf program analysis techniques. For many problems where classical techniques are insufficient, I have also developed new principled techniques. For instance, in order to verify safety properties of polynomial continuous dynamical systems, I developed a novel provably sound and complete deductive verification rule, as all existing rules were either unsound or incomplete.

1 Research Accomplishments

Over the last four years, I have worked on applying programming language techniques for improving the security of JavaScript web applications (thesis research), developing deductive verification techniques for hybrid and continuous dynamical systems, and developing constraint-solving techniques for synthesizing symbolic instruction encodings for processor instruction sets. What follows is a brief summary of my research in these areas.

1.1 Sandboxing Untrusted JavaScript

Most websites today incorporate untrusted third-party content in the form of advertisements, maps and social networking applications. Such content often consists of JavaScript code that accesses the page's Document Object Model (DOM) in order to manipulate features on the page. Since the DOM provides access to cookies, form fields, location bar, and other sensitive elements, untrusted JavaScript poses a significant security threat to the hosting page. For instance, a malicious ad on a page with a login form could use JavaScript to read login credentials from the form and send them to a remote server.

Websites presently combat this threat by filtering and rewriting untrusted JavaScript before placing it on the page. There are a number of such JavaScript "sandboxing" tools, including Facebook FBJS, Yahoo! ADSafe, Google Caja, and Microsoft WebSandbox. Despite their popularity, these mechanisms do not come with any rigorous specifications or guarantees. We therefore ask the following questions: *What are the security policies that these mechanisms try to enforce? Are these policies correctly enforced? How can we systematically design sandboxing mechanisms that correctly enforce these policies by design?* The rest of this section provides an overview of my work on answering these questions.

Operational Semantics for JavaScript. In order to reason formally about JavaScript programs, we developed a small-step operational semantics[1] for JavaScript, based on the 3^{rd} edition of the ECMA-262 Standard [2]. We call this language `ES3`. Unlike previous formalizations that focused on small core subsets of JavaScript, our semantics models the entire language as described in the standard. Our model could, therefore, be used for reasoning about JavaScript code "in the wild," which is crucial for security analysis.

Analysis of Facebook FBJS, Yahoo! ADSafe, and Google Caja. We reviewed the implementations of Facebook FBJS, Yahoo! ADSafe, and Google Caja and found that all of them implement a reference monitor using a combination of filtering and source-to-source rewriting of untrusted code, and wrapping of critical hosting page objects. We inferred three common security policies that these monitors try to enforce:
(1) *Hosting Page Isolation*: Certain security-critical objects (such as `window`, `document.cookie`) must be isolated from third-party components.
(2) *Inter-Component Isolation*: Third-party components must be isolated from each other.
(3) *Mediated Access*: Access to security-critical objects must always be mediated.
We initially tried to prove, using our semantics, that these three security policies are correctly enforced by the FBJS and ADSafe mechanisms. In trying to set up the correctness proof, however, we discovered exploitable

[1] Available online at `http://jssec.net`.

vulnerabilities with respect to all three policies in both FBJS and ADSafe. Due to the subtlety of these vulnerabilities, and others that might occur in similar systems, we decided to develop a principled approach to designing sandboxing mechanisms that enforce these policies.

Hosting-page Isolation. In a series of papers [6, 4, 3], we systematically designed language-based sandboxing mechanisms that enforce hosting page isolation on untrusted `ES3` code. Each mechanism was backed by a rigorous proof of correctness carried out using our operational semantics. The mechanisms varied in their level of restrictiveness over untrusted code, with the one in [3] being the least restrictive and comparable to FBJS. Although language-based sandboxing mechanisms have been developed previously in other contexts, designing them for JavaScript was challenging due to the lack of lexical scoping, lack of closure-based encapsulation, and the overall non-standard semantics of the language.

Inter-component Isolation. Inspired by Google Caja, in [5] we investigated object-capability methods for systematically enforcing inter-component isolation. We developed a language-based foundation for carrying out isolation proofs based on object-capability concepts and used it to design a capability-based sandboxing mechanism for enforcing inter-component isolation. This work was, to our knowledge, the first formal characterization of capability-safe programming languages.

Mediated Access. A standard technique for ensuring mediated access to security-critical objects is to hide the objects behind an API and allow untrusted code to interact only with the API. While the sandboxing mechanisms developed previously can be used for restricting untrusted code to an API, an important problem still remains: showing that the API safely *confines* all security-critical objects - that is, no interleaving of API calls ever leads to a reference to a critical object. Reasoning about all possible interleavings of API calls can only be done by statically analyzing the API implementation. Static analysis of `ES3` programs can be notoriously hard due to the various dynamic code generation features and the overall non-standard semantics of the language.

In December 2009, the ECMA standards committee released a 5^{th} edition of the standard that included a "strict mode" with a much more standard semantics than `ES3`. While the strict mode got rid of most static analysis hurdles, the issue of analyzing dynamically generated code still remained. In recent work [7], we tackled this issue by developing a sublanguage of strict mode JavaScript, called *Secure ECMAScript* (`SES`), that only allows dynamic code generation within statically declared scopes. We formulated the confinement problem for `SES` APIs as a "points-to" analysis problem and solved it using a standard Datalog based program analysis technique. We formally proved soundness of our technique and implemented it in the form of a fully automated tool called `ENCAP`.[2] Using `ENCAP`, we found a previously undiscovered confinement vulnerability in the Yahoo! ADSafe API, developed a fix for the vulnerability and subsequently verified confinement for the fixed API.

Overall Impact. A direct impact of this line of research has been in improving the security of real-world JavaScript sandboxing mechanisms by subjecting them to manual and automated analyses, and developing principled fixes to the problems we found. In most cases, our fixes were quickly adopted by the concerned vendors, further highlighting the benefits of this research methodology. The corner cases encountered while developing the operational semantics for JavaScript (`ES3`) have had an influence on the JavaScript standards committee (ECMA-262) and have guided some of the improvements made by strict mode JavaScript over the previous version. Furthermore, the operational semantics has also been used by other researchers as a foundation for developing JavaScript program analysis techniques. For instance, researchers at Imperial College London have used it to prove soundness of a program logic for JavaScript. Finally, we believe that our sublanguage SES has been able to hit the "sweet spot" between usability and analyzability, and in future we expect it to be useful for implementing security-critical code where strong guarantees are required. A proposal based on SES is currently under review by the ECMA committee for adoption within future versions of JavaScript.

1.2 Hybrid Systems Verification and Synthesis

Hybrid systems are physical systems with both continuous and discrete dynamics. A classic example is that of a thermostat that has an "on" and an "off" mode, with continuous dynamics governing the temperature within each mode and a discrete logic for switching between modes. In deploying a thermostat in safety-critical contexts, one might ask the questions: *Does the temperature always stay within 60F and 80F? Does the temperature always reach 70F?* In this work we aim to answer questions like these for the class of polynomial hybrid systems (systems where all dynamics are expressed using polynomial functions and all involved sets are semi-algebraic).

[2] `ENCAP` is open-source and is available at `http://code.google.com/EncapsulationAnalysis`.

In a series of papers [9, 8, 10, 11], we defined deductive rules for verifying and automatically synthesizing switching logics for safety and reachability properties of polynomial hybrid systems. Analogous to deductive program verification, the key idea is to verify safety by finding inductive invariants and reachability by finding inductive Lyapunov functions. The central challenge in developing this idea lies in defining the notion of "inductiveness" for continuous dynamics. We use ideas from control theory and differential geometry to define constraints for checking "inductiveness." The resulting rules are expressed as as $\exists\forall$ formula over the theory of reals (which is decidable), and are backed by proofs of soundness and completeness.

Deductive techniques for verifying hybrid systems are promising as they are sound and complete and unlike traditional approaches do not require solving the differential equations for each mode. In the future, I plan to develop deductive techniques for verifying more complex properties of hybrid systems such non-zenoness and fairness (across different modes).

1.3 Synthesizing Instruction Handler Encodings

Symbolic execution is a form of program analysis that involves executing a program with symbolic values rather than actual values. It forms a key component of most modern binary program analysis tools for test generation, verification, and malware analysis. Developing a symbolic execution engine for binary programs requires precise symbolic encodings (transfer functions) for each instruction. Such encodings are traditionally written by hand by reading the processor instruction manual, a process that is tedious (many instructions), error-prone (many corner cases), partial (not all instructions are usually covered) and imprecise (approximations are often used). In this work, we explored a radically different approach: *Can we synthesize the symbolic instruction encodings automatically?*

Building upon recent advances in automated synthesis, we recently [1] designed a novel template-based synthesis algorithm for automatically synthesizing symbolic instruction encodings (expressed as bit-vector constraints) from input-ouput samples, for a given (black-box) processor. Using the algorithm, we automatically synthesized symbolic encodings for over 500 x86 instructions (8/16/32-bits, outputs, EFLAGS). During this work, we also discovered several inconsistencies across x86 processors, errors in the x86 Intel spec, and bugs in previous manually-written x86 instruction handlers.

These results are promising and suggest that such automatic synthesis of instruction encodings may be possible for other instruction sets as well. In the future I plan to explore the ARM and x64 architectures, as well as floating point and SIMD instructions of x86. This could have significant impact as it would allow for automatically bootstrapping the construction of a symbolic execution engine for an instruction set.

2 Sample Future Directions

While the topics of my current research have significant potential for further exploration, I am also excited about researching new areas in programming languages and security in the future. Some projects that I have recently been interested in are as follows:

Defensive Consistency. In my previous research, I have only analyzed APIs under a single possible malicious client. Most web applications today are "web mashups" comprised of multiple third-party components. Difficult problems arise when we consider APIs that are subjected to multiple clients, some of which may be malicious. One such problem is checking that an API is *defensively consistent*, a problem that was first defined by Mark Miller in this doctoral thesis. Informally, an API is defensively consistent if it provides "good" service to all "good clients", irrespective of all malicious (non-good) clients. Here good clients could be defined as those that respect certain API usage rules and good service could be defined as some post-condition that the API guarantees. Note that defensive consistency is more general than conventional precondition-postcondition style correctness. In essence, defensive consistency amounts to checking that malicious clients cannot damage some internal API state such that the API stops providing good service to good clients. In the special case where all API usage rules can be defined as a pre-condition on the inputs, defensive consistency can be enforced using *input validation*. In the future I would like to formalize defensive consistency, and develop static analysis and run-time monitoring based techniques for enforcing it.

Automatic Specification Mining. Most applications today are essentially *extensions* of some underlying application platform. The platform provides services to the extension using a library or more generally an SDK.

For instance, web applications make use of the browser's DOM library, browser extensions use the extensions library, and mobile phone applications use the mobile platform's SDK. The specifications for such libraries and SDKs are typically documented only in English, and therefore are inherently ambiguous and incomplete. In order to reason precisely about the correct and secure functioning of applications, it is crucial to have formal specifications for the underlying libraries. In the absence of such specifications, most analysis techniques either choose to ignore the platform library or make worst case assumptions about it, compromising soundness in the former case and facing a large number of false positives in the latter.

In the future, I want to develop automated techniques for synthesizing rigorous, albeit partial, specifications of platform libraries. I plan to start with coarse specification problems: *What global state gets side-effected as a result of calling a library method? What data flows occur between the arguments and the return value of a library method?* Having such coarse specifications can be tremendously more useful than having nothing at all. In solving these problems, I plan to explore a combination of static and dynamic analysis techniques. Most library implementations come with large test suites. One possible direction would be to develop specifications for the particular code paths taken by the test runs, and then develop input preconditions that guarantee that only tested code paths are taken. I am optimistic that approaches like these could work due to the overall success of combined static and dynamic program analysis techniques like concolic execution, dynamic type inference and invariant mining.

Program Analysis Tools for JavaScript. I have a growing interest in developing better program analysis tools for JavaScript. Application developers can benefit immensely from automated test generation, type inference, and refactoring tools. While the browser vendors have invested in providing debugging and dynamic analysis tools for JavaScript, very little effort has been put into developing sophisticated static analysis tools. One reason for this is that `ES3` is poorly suited to static analysis. With the recent release of "strict mode" JavaScript and our proposed sublanguage `SES`, however, most of the `ES3`-specific static analysis hurdles have disappeared. Furthermore, I believe that the clean semantic properties of `SES` and the presence of tools such as `ENCAP` should serve as motivation for programmers to adopt this language. I would thus like to develop program analysis tools for `SES` that aid programmers in developing security-critical code. The lack of program analysis tools is a problem faced by almost all scripting languages that trade static typing with dynamic features in order to facilitate rapid prototyping. I expect the results obtained in developing program analysis techniques for JavaScript to also carry over to other scripting languages.

References

1. P. Godefroid and A. Taly. Automated synthesis of symbolic instruction encodings from i/o samples. to appear in Proc. of *Symposium on Programming Language Design and Implementation (PLDI)*, 2012.
2. S. Maffeis, J. C. Mitchell, and A. Taly. An operational semantics for JavaScript. In *Proc. of Asian Programming Languages Symposium (APLAS)*, 2008.
3. S. Maffeis, J. C. Mitchell, and A. Taly. Isolating JavaScript with filters, rewriting, and wrappers. In *Proc. of European Symposium on Research in Computer Security (ESORICS)*, 2009.
4. S. Maffeis, J. C. Mitchell, and A. Taly. Run-time enforcement of secure Javascript subsets. In *Web 2.0 Security & Privacy (W2SP)*, 2009.
5. S. Maffeis, J. C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2010.
6. S. Maffeis and A. Taly. Language-based isolation of untrusted Javascript. In *Proc. of Computer Security Foundations Symposium (CSF)*, 2009.
7. A. Taly, U. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra. Automated analysis of security-critical Javascript APIs. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2011.
8. A. Taly and A. Tiwari. Deductive verification of continuous dynamical systems. In *Proc. of International conference on Foundation of Software Technology and Theoretical Computer Science (FST&TCS)*, 2009.
9. A. Taly and A. Tiwari. Synthesizing switching logic using constraint solving. In *Proc. of International conference on Verification Model Checking and Abstract Interpretation (VMCAI)*, 2009.
10. A. Taly and A. Tiwari. Switching logic synthesis for reachability. In *Proc. of International conference on Embedded Software (EMSOFT)*, 2010.
11. A. Taly and A. Tiwari. Synthesizing switching logic using constraint solving. In *Proc. of International journal on Software Tools for Technology Transfer (STTT)*, 2011.