Building Models for Verification of Security Properties

Marius Minea

Politehnica University of Timișoara, Romania
marius@cs.upt.ro

Security is a classical example of a domain where errors in system design are frequent, yet often hard to find before the system is deployed. Thus, applying formal methods for verification is much needed. To do this, it is crucial to have system models that are accurate enough to account for the targeted flaws, yet employ the right abstractions to keep them tractable by model checking tools.

We discuss how to obtain such models for verifying security properties of web applications: extracting a model by analyzing the code, if available, as well as inferring a model from application behavior.

Model extraction from web application code Web applications are prone to well-known specific flaws such as cross-site scripting or SQL injection, but also to broken authentication and business logic errors which are harder to pinpoint and thus test against. Model checking with its exhaustive state-space exploration capability is a natural candidate to apply, and numerous model checkers for security protocol exist; however, they are mostly used with hand-written models. Moreover, the ability of the Dolev-Yao attacker to generate new messages from known parts necessarily leads to very large state spaces.

Thus, the challenge is to build a tool that automatically extracts models that accurately capture the application workflow, yet employ suitable abstractions to keep the models small. We discuss the jModex tool [3] built in the context of the SPaCIoS project [4] to extract models from web applications written using Java ServerPages (JSP). The resulting models can be analyzed by the model checkers of the AVANTSSAR platform [2].

jModex works as one might expect by backward traversal of the control flow graph for each Java method, tracking dataflow and path conditions to build an extended finite state machine. Crucially however, jModex is not a general-purpose Java model checker. It has semantic knowledge about the API for interacting with HTTP requests and exploits the typical structure of such applications (built around a server loop). It does not handle full-fledged Java (e.g., recursion, polymorphic calls); on the other hand, it provides a database model and has support for a subset of SQL queries, which is crucial for handling practical applications. jModex is configurable and extensible through user-specified abstractions, which allow to specify the semantics of certain methods in terms of its meta-model and thus adapt the abstraction level to the security property of interest. In effect, one can view jModex as a framework for building custom model extractors. Analyzing extracted models with the CL-AtSe model checker, we have demonstrated finding a bug in an open-source bookstore application.

Model inference from application behavior Recently, model-learning [5], based on Angluin's automata learning algorithm [1] has emerged as a practical method for error detection. It can be used to reverse-engineer system models, check protocol conformance with a specification, compare two implementations, etc.

For a web application, model learning is in essence a form of 'smart crawling'. It creates a 'page graph' of the application, where nodes are pages and edges are transitions (through form submission) or links between pages. Inferring a crawled (black-box) model can be useful since a source-extracted model may be imprecise or have unwanted implementation detail. Again, employing the suitable abstraction is crucial: pages which differ just in the values of dynamical elements (e.g. books in a bookstore) are deemed equal, as are links with the same target but different URL parameter values. Crawling stops when no new page representatives are found, their successors being similar to a given depth.

Comparing white-box and black-box models While not identical, models extracted from code (white-box) and inferred from exercising the application (black-box) should represent the same behaviors. Since crawling only follows the interface (links and forms) available to the user, behaviors allowed by the code but not found by crawling are potential workflow vulnerabilities.

By composing transitions in the code model to macro-transitions from receiving a request to sending a response, the models become compatible at transition level. A path in the white-box model with a path condition that does not satisfy any crawled path would be an undesired execution (e.g. authentication bypass). These can be checked by feeding path conditions to an SMT solver such as Z3. Thus, the existence of both types of model can be exploited in verification.

Acknowledgments This is joint work with Petru Florin Mihancea, who developed the jModex tool, and with former students Alex Gyori, Cristian Iuga and Csongor Mátyás-Barta in the European FP7 project no. 257876 SPaCIoS.

References

[1] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.
[2] A. Armando, W. Arsac, T. Avanesov, M. Barletta, A. Calvi, A. Cappai, R. Carbone, Y. Chevalier, L. Compagna, J. Cuéllar, G. Erzse, S. Frau, M. Minea, S. Mödersheim, D. von Oheimb, G. Pellegrino, S. E. Ponta, M. Rocchetto, M. Rusinowitch, M. T. Dashti, M. Turuani, and L. Viganò. The AVANTSSAR platform for the automated validation of trust and security of service-oriented architectures. In *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference*, pages 267–282, 2012.
[3] P. F. Mihancea and M. Minea. jModex: Model extraction for verifying security properties of web applications. In *Proceedings of the IEEE CSME-WCRE Software Evolution Week*, pages 450–453, 2014.
[4] SPaCIoS Project. Deliverable 5.5: Final Tool Assessment, 2014. Available at http://spacios.eu/deliverables/spacios-d5.5.pdf.
[5] F. Vaandrager. Model learning. *Communications of the ACM*, 60(2):86–95, 2017.