MOPS is a program analysis tool that uses formal verification techniques to find security bugs in software. The vulnerabilities targeted by this work involve “temporal safety properties,” which dictate the order of a sequence of security-relevant operations. An example of a temporal safety property is that a program that calls `chroot()` should then immediately call `chdir("/")` to change its working directory to the new root. A violation of such a property may mean that the program is vulnerable to attack, so for this reason, it is desirable to know whether a program obeys these safety properties.

Given formal models of the safety properties and of the program itself, MOPS uses standard model checking techniques to determine whether there is any execution path through a program that violates specified properties. Each safety property is modeled as a finite state automaton (FSA). In these models, the final states in an FSA are considered “bad” states that represent violations of the security property. Since finite state automata can be composed, it is easy to start with a set of small, simple security properties and combine them to obtain more complex properties. This encourages reuse, much like modularization of software itself. Each program to be checked for safety is modeled as a pushdown automaton (PDA). The states and transitions in the PDA represent control flow through the program.

The goal of MOPS, then, is to determine whether any possible trace in the program causes the FSA to reach a bad state. This is done by composing the FSA representing the security property and the PDA representing the program, and checking whether the language accepted by the resulting PDA is empty.

The paper gives an example of constructing a security model that describes the transition of privilege in a process in Linux 2.4.17. To construct the model, a user must first determine what kernel variables affect the security-relevant operations and then using this information to determine the states of the FSA. This step is done by hand. Then the user can run a state-space explorer to find the transitions among the states.

The authors evaluated MOPS by using it to find security bugs in several existing programs (wu-ftp, sendmail, and OpenSSH). Performance was measured using only the 53000-line Sendmail; MOPS took almost 2 minutes to parse the source files, 95 seconds for the model checking, and 300MB of memory for the computation.

This work was interesting and did have some good results. Since the authors used automata and model checking, two widely studied subjects, they were able to use well-known techniques to achieve their goals. Since the checking is static, there is no runtime overhead for using MOPS. MOPS is also sound. Soundness is hard to achieve in verification systems, and in many cases total soundness is not done in favor of practicality. For programs that satisfy the authors’ assumptions, MOPS can be a very useful tool. However, I think that some of their assumptions are too confining. For example, MOPS wouldn’t work with programs that have signal handling. Also, it does not work with multithreaded programs. A large number of programs in today’s computing environment are multithreaded, so this seems very limiting. Of course, a multithreaded program would cause the number of states in the PDA to explode (if this representation is even possible), so it would probably not be practical to use MOPS on those programs even if they tried to make it work.

Another concern I have about this paper is that I’m not sure if it’s really so easy to construct the PDA and FSA models. The authors imply that it’s not difficult, but it still does seem to be a largely manual task and therefore, prone to error. An error in the models could certainly result in false negatives. Finally, I think the evaluation presented in this paper was not that extensive; I would have liked to see some unknown bugs instead of just known ones, and tests of some more large programs in order to evaluate further the scalability of the system.