This paper presents MOPS, a formal approach for finding bugs in security-relevant software and verifying their absence. The idea behind MOPS is to map the security property to a Finite State Automaton, and the program as a Pushdown Automaton. MOPS then use model checking to determine whether certain states representing violation of the security property in the FSA are reachable in the PDA.

One advantage of MOPS is that it fully supports interprocedural analysis, as opposed to local analysis which only keeps track of states and variables in a single procedure. This is useful, as shown in the example where `drop_privilege()` forgets to drop privilege in case of error, and return to the main function which then calls `execl()`.

Using PDA to model the program has its pros and cons. Since there are efficient algorithms to compute the intersections of a PDA and an FSA, and by approximating the program traces (which is in general an uncomputable set, thus making deciding whether the two languages’ intersection is null undecidable) to a context free language, the problem becomes decidable.

However, this also means that some imprecision is introduced by approximating the program trace to a PDA. This can be attributed to the fact that data flow is not modeled, the checker will go into paths that are infeasible and report them as errors.

MOPS has the property that it is sound – it will make mistakes by making false alarms but will not overlook a real violation. This can be very useful when doing an audit, but for general debugging purpose, having a lot of false positives will limit its usefulness, for the programmer will not have the patience to look though say a hundred false positives to find a real error. The paper did not give sample numbers of the ratio of real errors to false alarms, so it is not obvious how useful MOPS is when used in real applications. In contrast, the meta-level compilation paper gave solid numbers on the number of false positives and the number of real errors found. If the MOPS paper also present these data, say, with the result of finding the vulnerability in `wu-ftpd` 2.4 beta 11, it will give readers a better picture of how useful the software is.

As for the data flow insensitive problem, the authors suggest doing a rudimentary data flow analysis by encoding data values into the security model. This is a compromise between the scalability (if all data flow is to be kept track of, the program will become prohibitively slow and complex) and number of false positives. I think this is a good way to give the user some control in the number of false positives. Say there are a lot of errors reported but they are all caused by going down a path that is dependent on a boolean. By introducing the boolean into the security model, all these irrelevant errors may be eliminated, while adding some complexity to the model. Therefore it is up to the user to decide which false positives are too annoying to be worth getting rid of and which do not appear so often as to worth adding to the security model.
Another problem that MOPS faces is signal handling, since \textit{longjmp} disrupts the flow of the program and currently it is not handled in MOPS, but the authors plan to add transitions for signal handlers automatically in future version of MOPS. Currently they need to modify the program by hand, and this may introduce some problems, say missing some of the signal handler statements, and create more human careless mistake problems that MOPS is set out to solve in the first place.

Overall, MOPS is a good alternative for meta-level compilation, and there is the tradeoff between scalability and the number of false positives. MOPS aims at not missing a single error while MC aims to strike a good balance between them. I can imagine combining MC and MOPS to get a better tool in identifying bugs in programs.