ABSTRACT

In recent years, binary code analysis, i.e., applying program analysis directly at the machine code level, has become an increasingly important topic of study. This is driven to a large extent by the information security community, where security auditing of closed-source software and analysis of malware are important applications. Since most of the high level semantics of the original source code are lost upon compilation to executable code, static analysis is intractable for, e.g., fine-grained information flow analysis of binary code. Dynamic analysis, however, does not suffer in the same way from reduced accuracy in the absence of high-level semantics, and is therefore also more readily applicable to binary code. Since fine-grained dynamic analysis often requires recording detailed information about every instruction execution, scalability can become a significant challenge. In this thesis, we address the scalability challenges of two powerful dynamic analysis methods whose widespread use has, so far, been impeded by their lack of scalability: dynamic slicing and instruction trace alignment. Dynamic slicing provides fine-grained information about dependencies between individual instructions, and can be used both as a powerful debugging aid and as a foundation for other dynamic analysis techniques. Instruction trace alignment provides a means for comparing executions of two similar programs and has important applications in, e.g., malware analysis, security auditing, and plagiarism detection. We also apply our work on scalable dynamic analysis in two novel approaches to improve fuzzing — a popular random testing technique that is widely used in industry to discover security vulnerabilities. To use dynamic slicing, detailed information about a program execution must first be recorded. Since the amount of information is often too large to fit in main memory, existing dynamic slicing methods apply various time-versus-space trade-offs to reduce memory requirements. However, these trade-offs result in very high time overheads, limiting the usefulness of dynamic slicing in practice. In this thesis, we show that the speed of dynamic slicing can be greatly improved by carefully designing data structures and algorithms to exploit temporal locality of programs. This allows avoidance of the expensive trade-offs used in earlier methods by accessing recorded runtime information directly from secondary storage without significant random-access overhead. In addition to being a standalone contribution, scalable dynamic slicing also forms integral parts of our contributions to fuzzing. Our first contribution uses dynamic slicing and binary code mutation to automatically turn an existing executable into a test generator. In our experiments, this new approach to fuzzing achieved about an order of magnitude better code coverage than traditional mutational fuzzing and found several bugs in popular Linux software. The second work on fuzzing presented in this thesis uses dynamic slicing to accelerate the state-of-the-art fuzzer AFL by focusing the fuzzing effort on previously unexplored parts of the input space. For the second dynamic analysis technique whose scalability we sought to improve — instruction trace alignment — we employed techniques used in speech recognition and information retrieval to design what is, to the best of our knowledge, the first general approach to aligning realistically long program traces. We show in our experiments that this method is capable of producing meaningful alignments even in the presence of significant syntactic differences stemming from, for example, the use of different compilers or optimization levels.