Grammar Inference via Dynamic Taint Tracing

Pavel Zakopaylo, ANU

Supervisors: Steve Blackburn, Anthony Hosking, Michael Norrish, Shane Magrath

Motivation

- Automated security analysis of software applications (i.e. finding bugs; think fuzzing)
- Want to describe the input data format;
- Application is given as a compiled binary – no source code or debugging symbols
- Analysis of data flow in memory is important to achieving these aims

Prior Work

- TUPNI – Microsoft Research
- Taint Tracing as Data Flow Analysis
- Taints :: memory location → { file offset }

  … at any point during execution

  … e.g. 0x7fff…4 → { 0, 1, 4 }
- TUPNI paper does not precisely describe how these taints are determined, and cites papers that are similarly vague

My Taint Tracer

- Built using Intel‘s PIN
- Intercepts Linux syscalls on IA-32 and Intel-64 architectures
- Inserts update to taint database after each (relevant) instruction
- At attempt to replicate the undocumented features of TUPNI

Implementation

- PIN‘s API provides some functionality to determine which memory locations / registers are read / written
- Mostly dealing with edge cases
- e.g. PUSH, POP affect RSP deterministically, so do not spread taint
- Toy parsers written as „unit tests“

Performance

- Most (non-TUPNI) work on taint tracing cites „negligible“ performance overhead
- Most likely b/c they only track *whether* a location is tainted (i.e. memory overhead <= 12.5%), as opposed to *which* bytes in the input are responsible for the taint
- My testing shows **orders of magnitude** more memory & CPU usage
- Performing an O(log N) operation after every instruction is **slow**

// TODO

- Mitigate performance issues, to the point where it‘s at least usable
- (e.g. try to compress taint data a la Virtual Memory, try to use O(1)-search structures)
- Make tool connect to an actual grammar inference system
- (Or to an architecture-independent format as originally suggested by Shane)
- More special cases
- Replicate TUPNI (!!)

Questions?