Static and Dynamic Analysis of ARM Binaries

Mark C. Reynolds
Research Fellow
markreyn@bu.edu
Hariri Institute of Computing, Boston University
Problem Statement

• Find security vulnerabilities before they are exploited
  • On average 10 months elapses between the deployment of malicious code and its detection [1]
  • Some malware has remained unexposed for as long as 3 years
  • There is a thriving underground marketplace for 0-day vulnerabilities
• Standard antivirus software is inadequate at detecting 0-days
  • Signature based approaches are 99%+ on known malware, but only 5% on novel malware
  • Anomaly detection/machine learning is at least 95% accurate on known malware, but only 30-40% on novel malware
  • False positives (up to 5% of all detections) can require a human expert to disambiguate
  • Can be defeated by several known techniques
  • Very little effort for ARM to date
• Malware construction kits are inexpensive and readily available ($300 for the Dendroid kit for Android)
  • Machine code expertise is no longer required to create malware

1. Forbes; Andy Greenberg; Hackers exploit ‘zero-day’ bugs for 10 months on average before they’re exposed; published 16 Oct. 2012
Static and Dynamic Analysis

Static and Dynamic Analysis

Binary

Static Model or Theory

Model Checker or Proof System

Counterexamples

Proofs

Null Results

Emulated Execution

Memory Reference Checker

Positive Inferences

Negative Inferences

Native Execution
The Optimist’s Approach

• Existing malware detection attempts to identify “bad” software
• Our approach defines rules that must be obeyed by “good” software
  • Rules are written in a modeling language
  • Given a piece of software-under-test, it can be formally verified or formally refuted using a model checker or proof system
  • For finite model checkers (such as Alloy), a counterexample is always a vulnerability, but formal verification only applies to a given search depth
    • More compute power => deeper search => greater confidence
• The rules define what is “good”
  • The semantics of goodness may give unexpected results. For example, a program that deletes all your files may be classified as benign at the level of machine code, but malicious at the application level
    • Rules and models should be composable
Modeling Languages

• The Alloy finite model checker is fully supported
  • Alloy++ is fully supported

• The Isabelle proof system is mostly supported

• The UPPAAL modeling language for systems with real-time constraints is being developed
Success Stories

- **Java**
  - The Java analyzer operates on JVM bytecodes. All rules in the public specification are supported. All instructions, including the new “invokedynamic” are supported.
  - The Java analyzer has detected four 0-days several months before official disclosure

- **Flash**
  - The Flash analyzer operates on ActionScript (ABC) embedded with the Flash file
  - Coverage is approximately 75% of the ABC language
  - The Flash analyzer has detected two 0-days months before official disclosure

- **Android**
  - The Android analyzer provides full coverage for all Dalvik VM instructions and all rules in the specification
  - The Android analyzer has identified a critical flaw in a very popular application. A simple ~ 400 byte payload can crash your Android device without any human interaction. This is still not patched as of today (10/1/2014).
Embedded Software: ARM

- Effort began August 2013
- The ARM analyzer reached beta status in July 2014
  - All instructions that do not read/write memory have been captured
  - All rules related to such instructions have been captured
- JVM, ABC and DEX do not read/write memory, only virtual registers and a virtual stack. ARM machine code does
  - The existing model framework needed to be enhanced to provide a “probabilistic memory model”
  - The current implementation can catch many questionable memory accesses
  - More work is needed to completely model the entire ARM instruction set
Components

• Instruction Definition File (IDF)
  • Rules related to individual instructions
  • Written in a DSL
  • Example (JVM double add)
    Instruction 99 dadd
    Fromstack DoubleLow DoubleHigh DoubleLow DoubleHigh
    Tostack DoubleLow DoubleHigh
    Condition DOverflow NAN
    EndInstruction

• Generic model
  • Rules unrelated to specific instructions
  • Written in the modeling language
  • Example (Alloy, “never jump into the middle of an instruction”)
    assert InstructionTransfer_abs_u {
      all ins: Instruction, u: ins.ubt | one bti: Instruction { bti.map = u }
    }
Components Continued

- Translator
  - Executable written in Java or C++
  - Reads software-under-test, writes model initializers
  - Example (JVM + Alloy)
    ```
    ./sics one.jar
    ```
  - Creates the file one.als, an Alloy file that initializes all the Alloy relations defined in the generic model files jvm_before.als and jvm_after.als
  - This output file is known as the instance model
  - When the generic model and the instance model are combined, a complete model/theory in the selected modeling language is obtained. This combined model is specific to the software-under-test

- The IDF, the generic model and the translator are always used
Reasoning about Memory Accesses

• It does not seem computationally feasible to track all memory accesses through either static or dynamic analysis
• Therefore, we would like to capture statements that are probably true, for a specific definition of “probably”
• For example, we would like to encode the following observation as a probabilistic statement about memory accesses:
  • “After 1000 observations all accesses to the memory range [0xE0000,0xF0000) are read operations that read four bytes at a four-byte aligned address”
  • Therefore, with 10 bit certainty we claim that:
    • Any access to this range must be a read
    • Offset % 4 == 0
    • Len == 4
Embedded Components

• Probabilistic Memory Model Templates (PMMT)
  • A set of probabilistic rewriting rules
  • Written in Standard ML

• The Memory Monitor and Memory Shim (M3S)
  • Gleans addresses that can be extracted from static analysis (address literals)
  • Consumes the PMMT
  • Monitors execution of software-under-test
    • Inserts a shim (native code trampoline) that reports on memory accesses to the MM
    • Reports on rule violations above a given threshold of certainty; for example if the threshold=10 bits, and a single byte read of address 0xE0004 is detected, then a violation of the corresponding rule occurs with a certainty of k=10 (bits)

• Written in standard ML
• Very much a work in progress
System Architecture

Software Under Test

Translator

IDF

Generic Model

Instance model

Segment Engine

Segments

Memory Shim

Memory Monitor

PMMT

Cloud

Results

Model Checker

Results

Vulnerability Report

Synthesizer
Demo

- Good.elf: benign ARM code
  passes Alloy check and MM

- Bad.elf: malicious ARM code
  illegal instruction transfer
  Alloy counterexample, passes MM

- Ugly.elf: suspicious ARM code
  read(0xE0004, 1)
  passes Alloy check, fails MM with k=10