The Malware Binary

- **Why is analyzing malware hard?**
  - Packed code: the binary’s malicious code is not generated until runtime
  - Self-modifying code: overwrites can change the code’s behavior
  - Obfuscated code: what code is statically visible is hard to analyze
  - These techniques are pervasive! 90% of malware is analysis-resistant

- **Unpacking loop**
  - An unpacking loop decompresses or decrypts the hidden code at runtime

- **Code that will be overwritten**
  - The code that is statically present may be modified before or after it executes

- **Hidden code bytes**
  - Hidden code is compressed or encrypted and looks like random data

Dyninst Analysis of Binary

- **Control Flow Graph**
  - Address 0x401000: there’s nothing here at startup!

- **Static parsing**
  - Dyninst initially generates a CFG for all visible code

- **Obfuscated control transfer analysis**
  - We monitor two types of control transfer instructions:
    - Indirect transfers since we can’t statically determine where they go
    - Static transfers into empty or uninitialized regions

Analyzing Conficker

- **Conficker generates its malicious code at runtime; we use Dyninst to analyze its static and dynamic code.**

  - **Code coverage of basic blocks**
    - We efficiently obtain code coverage information for Conficker A by instrumenting all of its basic blocks and removing instrumentation that has executed.

  - **Instrumentation**
    - We find new code by instrumenting control transfers to determine whether the target address is in a new code region

  - **Modified code**
    - We track code writes by write-protecting code pages and capturing the resulting exceptions

  - **Runtime parsing**
    - Dyninst parses unpacked code regions just before they execute
    - Candidate control transfers to other unpacked regions are instrumented

  - **Obfuscated control transfer analysis**
    - We monitor two types of control transfer instructions:
      - Indirect transfers since we can’t statically determine where they go
      - Static transfers into empty or uninitialized regions

Before Execution

- **Before Execution**
  - **Memory image**
    - At runtime the unpacking loop unpacks hidden code into the address space
    - But where? We want to find the code before it executes
  - **Address Space**
    - The program’s static code has been replaced by modified code
  - **Unpacked code**
    - The code must be unpacked before it can be executed, though it could be unpacked a piece at a time
    - At some point a control transfer instruction passes execution to an unpacked code region
  - **File**
    - Hidden code decompresses or decrypts the hidden code at runtime
  - **Unpacking loop**
    - An unpacking loop decompresses or decrypts the hidden code at runtime
  - **Code that will be overwritten**
    - The code that is statically present may be modified before or after it executes
  - **Hidden code bytes**
    - Hidden code is compressed or encrypted and looks like random data

During Execution

- **During Execution**
  - **Memory image**
    - At runtime the unpacking loop unpacks hidden code into the address space
    - But where? We want to find the code before it executes
  - **Address Space**
    - The program’s static code has been replaced by modified code
  - **Unpacked code**
    - The code must be unpacked before it can be executed, though it could be unpacked a piece at a time
    - At some point a control transfer instruction passes execution to an unpacked code region
  - **File**
    - Hidden code decompresses or decrypts the hidden code at runtime