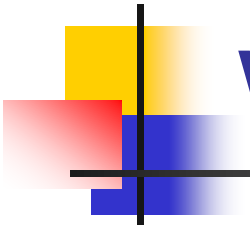
Pragmatic Approaches with COTS

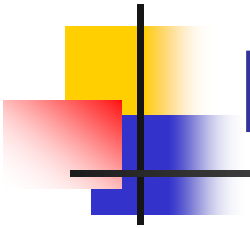
Reverse Engineering for the Detection of Undesirable Functionality

WISDOM SOFTWARE, LTD.



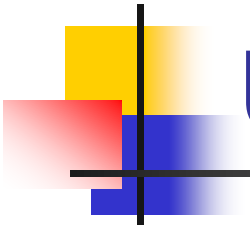
Wisdom Software, Ltd.

- Small Business- Incorporated in Virginia
- Business Interests
  - Formal Methods research and application development
  - Software Certification
  - System Safety Analysis
- Customer Base
  - NSA
  - US Army – WRAIR
  - FDA



Research Objectives

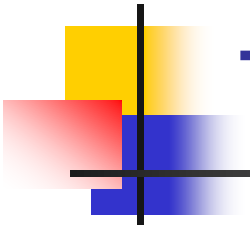
- Evaluate the practicality of using reverse engineering (RE) to identify undesirable functionality (UF)
- Document and implement a practical (formal methods) process for RE
- Examine implications for addressing system security and safety



Undesirable Functionality

Definition

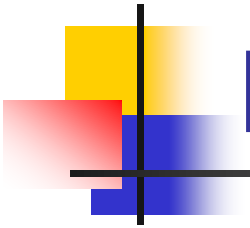
- permits non-secure access
- exports sensitive information
- damages computer or network operations
- modifies other unrelated systems or resources
- otherwise compromises system security



The UF Problem

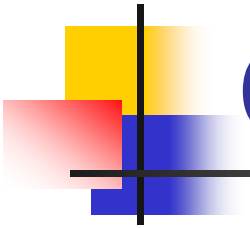
- There is potentially an infinite set of unique instances of UF
- Each instance might be implemented in many ways
- Implications
  - A general solution is needed
  - Functionality must be examined in context

Finding Undesirable Functionality

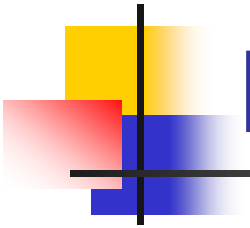


- Pattern matching is insufficient
  - Only useful if you already know specifically what you are looking for
  - Does not address variations on implementation
  - Does not identify unknown undesirable functionality
- The alternative: Examining functional behavior in context

Examination of Behavior in Context

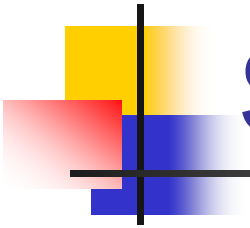


- Context is based on the scope of information abstracted from the software
  - Context of definition
  - Context of use
- Obvious behavior
  - Local function definition
  - Short stimulus sequence trigger
- Not so obvious behavior
  - Function definition is distributed
  - Long stimulus sequence trigger



Reverse Engineering Process

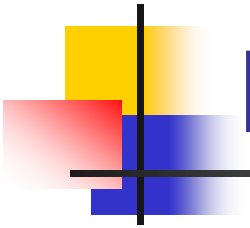
- Segment the software into functional blocks
- Abstract the program function for each segment
- Compose the abstracted segment functions
- Generate the Legal Sequence Table from the composed segment function



Segmentation

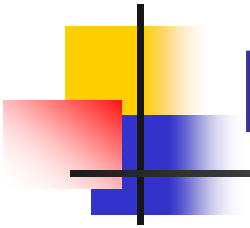
- Segmentation divides the software into functional blocks
- A segmentation graph shows the hierarchical relationship of the segments
- Each segment represents a context of definition

```
static void free_proc_chain(struct process_chain *procs)
{
    struct process_chain *p;
    int need_timeout = 0;
    int status;

    if (procs == ((void *)0) )
        return;

# 2666 "alloc.c"

    for (p = procs; p; p = p->next) {
        if (waitpid(p->pid, (int *) 0, 1 ) > 0) {
            p->kill_how = kill_never;
        }
    }


    for (p = procs; p; p = p->next) {
        if ((p->kill_how == kill_after_timeout)
            || (p->kill_how == kill_only_once)) {

            if (kill( p->pid ,   15  )  != -1)
                need_timeout = 1;
        }
        else if (p->kill_how == kill_always) {
            kill(p->pid, 9 );
        }
    }

    if (need_timeout)
        sleep(3);

    for (p = procs; p; p = p->next) {

        if (p->kill_how == kill_after_timeout)
            kill(p->pid, 9 );

        if (p->kill_how != kill_never)
            waitpid(p->pid, &status, 0);
    }
}
```



Function Abstraction

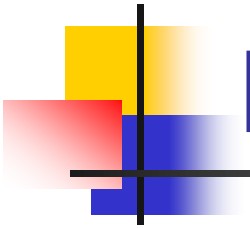
- Function abstraction is the process of deriving a state machine for a segment
- A state machine is abstracted for each segment in the system being analyzed
- The state machine is represented in a tabular format for each segment



Function Composition

- Abstracted state machines for related segments are combined by function composition
- A loop unrolling algorithm is employed to eliminate dynamic looping or recursion in the software
  - Resulting definition of behavior is an approximation
  - Accuracy increases as the number of cycles unrolled is increased

Segment 82			
Stimulus	**Current Condition**	**State Update**	**Response**
Invoke	procs == null		return terminate
Invoke	procs != null		waitpid(procs->pid, (int*) 0, 1)
waitpid(procs->pid, (int*) 0, 1) > 0	procs!= null	procs->kill_how = kill_never	terminate
waitpid(procs->pid, (int*) 0, 1) ≤ 0	procs!= null (procs->kill_how == kill_after_timeout OR procs->kill_how == kill_only_once)		kill(procs->pid, 15)
waitpid(procs->pid, (int*) 0, 1) ≤ 0	procs!= null procs->kill_how == kill_always		kill(procs->pid, 9) waitpid(procs->pid, &status, 0) terminate
waitpid(procs->pid, (int*) 0, 1) ≤ 0	procs!= null procs->kill_how != kill_after_timeout procs->kill_how != kill_never procs->kill_how != kill_only_once procs->kill_how != kill_always		waitpid(procs->pid, &status, 0) terminate
waitpid(procs->pid, (int*) 0, 1) ≤ 0	procs!= null procs->kill_how == kill_never		terminate
kill(procs->pid, 15) == -1	procs!= null procs->kill_how == kill_after_timeout		kill(procs->pid, 9) waitpid(procs->pid, &status, 0) terminate
kill(procs->pid, 15) == -1	procs!= null procs->kill_how == kill_only_once		waitpid(procs->pid, &status, 0) terminate
kill(procs->pid, 15) != -1	procs!= null procs->kill_how == kill_after_timeout		sleep(3) kill(procs->pid, 9) waitpid(procs->pid, &status, 0) terminate
kill(procs->pid, 15) != -1	procs!= null procs->kill_how == kill_only_once		sleep(3) waitpid(procs->pid, &status, 0) terminate

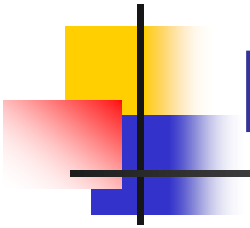


Legal Sequence Table

The composed segment table is processed to derive the legal stimulus sequences

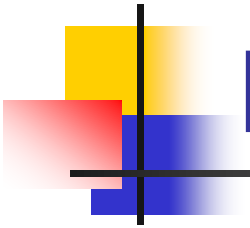
- This identifies behavior in context of use
  - Series of responses generated for each sequence
  - Series of state updates generated for each sequence
  - For lower level segments - possible conditions under which a sequence may occur
- Gives the analyst a clearer picture of the actions being performed by the segment

Segment 82 Legal sequences			
Sequence	**Conditions**	**State Updates**	**Responses**
Invoke	procs == null		return terminate
Invoke waitpid(procs->pid, (int*) 0, 1) > 0	procs != null	procs->kill_how = kill_never	waitpid(procs->pid, (int*) 0, 1) terminate
Invoke waitpid(procs->pid, (int*) 0, 1) ≤ 0 kill(procs->pid, 15) == -1	procs!= null procs->kill_how == kill_after_timeout		waitpid(procs->pid, (int*) 0, 1) kill(procs->pid, 15) kill(procs->pid, 9) waitpid(procs->pid, &status, 0) terminate
Invoke waitpid(procs->pid, (int*) 0, 1) ≤ 0 kill(procs->pid, 15) == -1	procs!= null procs->kill_how == kill_only_once		waitpid(procs->pid, (int*) 0, 1) kill(procs->pid, 15) waitpid(procs->pid, &status, 0) terminate
Invoke waitpid(procs->pid, (int*) 0, 1) ≤ 0 kill(procs->pid, 15) != -1	procs!= null procs->kill_how == kill_after_timeout		waitpid(procs->pid, (int*) 0, 1) kill(procs->pid, 15) sleep(3) kill(procs->pid, 9) waitpid(procs->pid, &status, 0) terminate
Invoke waitpid(procs->pid, (int*) 0, 1) ≤ 0 kill(procs->pid, 15) != -1	procs!= null procs->kill_how == kill_only_once		waitpid(procs->pid, (int*) 0, 1) kill(procs->pid, 15) sleep(3) waitpid(procs->pid, &status, 0) terminate
Invoke waitpid(procs->pid, (int*) 0, 1) ≤ 0	procs != null procs->kill_how == kill_always		waitpid(procs->pid, (int*) 0, 1 kill(procs->pid, 9) waitpid(procs->pid, &status, 0) terminate
Invoke waitpid(procs->pid, (int*) 0, 1) ≤ 0	procs!= null procs->kill_how != kill_after_timeout procs->kill_how != kill_never procs->kill_how != kill_only_once procs->kill_how != kill_always		waitpid(procs->pid, (int*) 0, 1) waitpid(procs->pid, &status, 0) terminate
Invoke waitpid(procs->pid, (int*) 0, 1) ≤ 0	procs!= null procs->kill_how == kill_never		waitpid(procs->pid, (int*) 0, 1) terminate



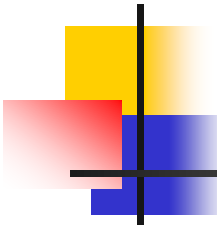
Results Analysis

- Behavioral description of the program function
  - State machine definition
  - Legal Sequence Table
- Potentially large volume of data to address
  - Prioritization – current research
  - Assertion tests – proposed research
  - Semantic analysis – proposed research



Prioritization

- Prioritization is being investigated to direct analyst attention to segments that have a higher likelihood of containing UF
- Based on a cursory examination of attributes
  - Segment level
  - State machine level
  - Legal sequence level
- Generates a weighted score for each segment that reflects a potential for or sensitivity to UF



Ongoing Research

- Refinement of the prioritization weighting scheme
  - Optimal weighting values for measured attributes
  - Additional attributes to examine
- Refinement of Abstraction process
  - Improved automation support
- Results Analysis and Presentation