Automated Program Behavior Analysis

Stacy Prowell

sprowell@cs.utk.edu

March 2005

SQRL / SEI

Motivation: Semantics

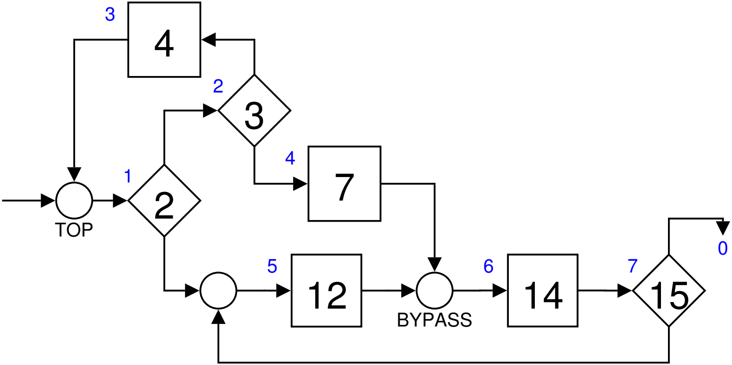
Development: Most engineering designs are subjected to extensive analysis; software is typically not.

Testing: Testing focuses heavily on flaws in the design, not on incorrect assumptions during design.

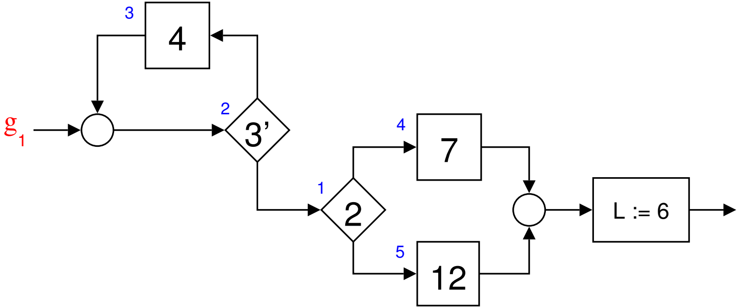
Analysis: The function of existing systems is not known with sufficient fidelity to make engineering decisions.

Motivation: Complexity

Large programs contain a huge number of execution paths, some of which may violate security or safety properties. Programmers cannot understand them all; they typically understand the main flow of the program and a few exceptional cases.



Every program is composed of a finite collection of structures, each of which implements a function from inputs to outputs.



Motivation: Exploit Structure

The function of each structure can be determined (extracted) based on rules for the particular structure (its functional semantics). These can be composed in a stepwise fashion to determine the overall program function.

Motivation: Expose Behavior

It is even hard to understand simple but "clever" programs.

```
void do_blink() {
    if (*BLINK) {
        *BLINK--;
        if (*BLINK <= 0) {
            *BLINK = 10;
            *SPEED_DISPLAY = 0xFF;
            return;
        } else if (*BLINK <= 5) return;
    }
    *SPEED_DISPLAY = *SPEED;
}
```

Motivation: Expose Behavior

The extracted function reveals what is happening.

$$\begin{aligned}&(\, b = 1 \implies b, s{:=}10, \mathrm{0xFF}\\&|\, 1 < b \leq 6 \implies b, s{:=}b-1, s\\&|\, b > 6 \implies b, s{:=}b-1, \mathbf{S}\\&|\, b = 0 \implies b, s{:=}b, \mathbf{S}\,).\end{aligned}$$

When `*BLINK` is one, the display is blanked and `*BLINK` is set to ten. Now, with `*BLINK` set to ten, the $b > 6$ rule takes over the next time through, and the display is immediately set to the speed. Thus the display is blanked for only 1⁄10 of a second, not for 1⁄2 of a second, as desired.

Basic Idea

Given an arbitrary program, generate the **program function** via function extraction.

Program $\rightarrow$ Program Function

Transform one specification (procedure) into another (procedure-free).

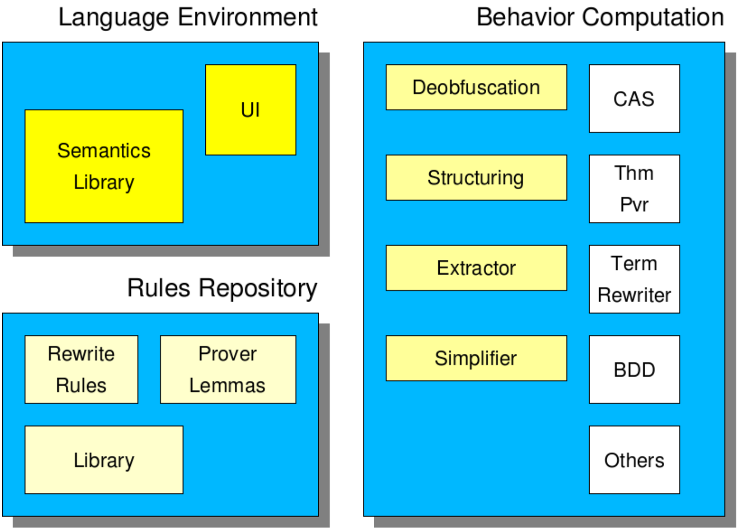
Perform this transformation in a way which is:

- Mathematically correct; avoid approximations
- Interactive; rely on knowledgeable users
- Extensible; provide ways to improve the extractor output

Even if the task is only partially successful, useful information is obtained.

Provide a **platform** for analysts and developers to use which supports reasoning about program function in a mathematically correct way.

Architecture of the System



Example: Source

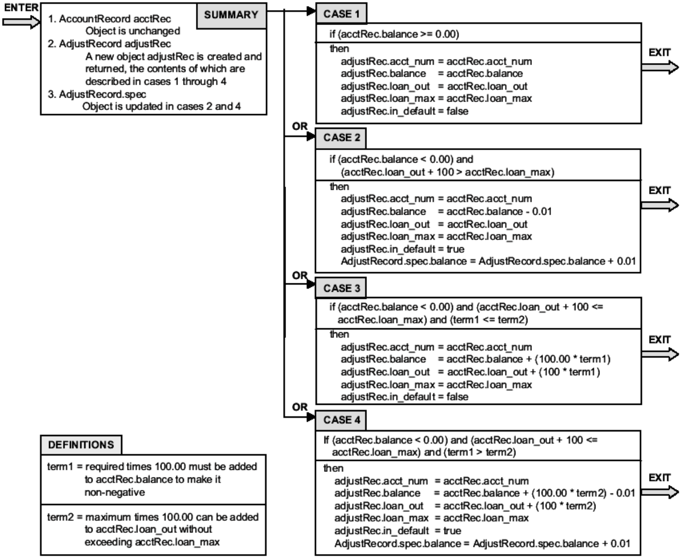
```
public class AccountRecord {
   public int acct_num;
   public double balance;
   public int loan_out;
   public int loan_max;
} // end of AccountRecord

public class AdjustRecord
extends AccountRecord {
   public boolean in_default;
   public static AdjustRecord spec;
} // end of AdjustRecord

public static AdjustRecord classify_account
(AccountRecord acctRec) {
   AdjustRecord adjustRec = new AdjustRecord();
   adjustRec.acct_num = acctRec.acct_num;
   adjustRec.balance = acctRec.balance;
   adjustRec.loan_out = acctRec.loan_out;
   adjustRec.loan_max = acctRec.loan_max;
   while ((adjustRec.balance < 0.00) &&
         ((adjustRec.loan_out + 100) <= adjustRec.loan_max)) {
      adjustRec.loan_out += 100;
      adjustRec.balance += 100.00;
   }
  adjustRec.in_default = (adjustRec.balance < 0.00);
  if (adjustRec.balance < 0.00) {
      adjustRec.balance -= 0.01;
      AdjustRecord.spec.balance += 0.01;
   }
   return adjustRec;
}
```

(From Pleszkoch and Linger 2004.)

Example: Behavior Catalog



Example: Exploited Code

CASE 2

```
if (acctRec.balance < 0.00) and
   (acctRec.loan_out + 100 > acctRec.loan_max)
then
   adjustRec.acct_num = acctRec.acct_num
   adjustRec.balance    = acctRec.balance - 0.01
   adjustRec.loan_out   = acctRec.loan_out
   adjustRec.loan_max = acctRec.loan_max
   adjustRec.in_default = true
   AdjustRecord.spec.balance = AdjustRecord.spec.balance + 0.01
```

CASE 4

```
If (acctRec.balance < 0.00) and (acctRec.loan_out + 100 <=
  acctRec.loan_max) and (term1 > term2)
then
   adjustRec.acct_num  = acctRec.acct_num
   adjustRec.balance    = acctRec.balance + (100.00 * term2) - 0.01
   adjustRec.loan_out   = acctRec.loan_out + (100 * term2)
   adjustRec.loan_max  = acctRec.loan_max
   adjustRec.in_default  = true
   AdjustRecord.spec.balance = AdjustRecord.spec.balance + 0.01
```

Components

- Deobfuscation
- Structuring

- Function Extraction
- Simplification

Deobfuscation

Idea

Rewrite the program flow to eliminate computed addresses (starpoints); transform these into case statements.

Combine precondition / postcondition analysis with execution chart analysis (H-Chart). Very similar to value set analysis.

- Detect false computed jumps, dead code, and computed constants.
- Transform computed jumps into case statements.

Benefit

Eliminate unreachable code, simplify program flow, expose indirect references.

Deobfuscation: Pointers

- Pointers are typically not used arbitrarily; often they are initialized and never changed.
- It may be possible to determine that a pointer is bounded in some way; a given range, or given strides.
- Where possible, convert pointers to case statements.

Structuring

Idea

Rewrite arbitrary program flow as structured program flow, using while, if-then-else, and sequence.

Rewrite program flow, possibly using label-structures where necessary.

Benefit

Provide a structured view of the program (annotated with or linked to the original source) through which analysts can navigate. This will be the central interface to the system.

Extraction

Idea

For each structure allowed by program structuring, determine the program function.

- Use function composition to generate overall function from component function.
- Limited number of structures simplifies the problem.

Benefit

Structures in the program can be annotated with program functions. The resulting program function can be put in the database or queried via a theorem prover or model checker.

Extraction: Loops

There is no general theory for loop abstraction.

- It is believed that a large number of loops can be recognized by pattern: count up, count down, copy memory, clear memory, search, etc. New patterns can be added based on analysis of programs.
- In some cases loop functions can be deduced by using quantifiers and rewrite rules.
- Discover loop function by iteratively guessing the loop invariant.
- If none of these works, write the loop as a recursive expression; perhaps the user will recognize and add the pattern to the database.

Simplification

Idea

Rewrite the extracted functions in referentially-transparent ways to simplify their presentation to the user, or to reduce the work done during extraction and composition.

- Term rewriting system
- Dedicated simplifiers (arithmetic, BDD)
- Library of known functions

Benefit

Users can add patterns which are specific to the domain of the program being studied.

Simplification: Other Tools

- Theorem provers (ACL2, PVS, ...)
- Model checkers
- Binary decision diagrams
- General term rewriting systems (Omega, Maude, ...)
- Computer algebra systems (Maple, Axiom, ...)

Abstraction

Idea

Hide details of inputs and outputs.

A “banking system” has certain characteristics which can be abstracted in a referentially-transparent manner, such as deposit, withdrawal, overdraft, etc. These have slightly different implementations in different programs.

Benefit

Further simplification.

Use Patterns

Discovery:

- Given a program, does the program's behavior catalog reveal any malicious activity?
- Does the program correctly implement the stated function?

Maintenance:

- Are two implementations the same after refactoring?
- Does a maintenance change preserve the function modulo the change?

Status

The basic decompilation, deobfuscation, structuring, extraction, and simplification systems have been developed and are being tested now for Intel bytecodes.

Work is underway now on loop recognition and extraction.

Collaboration

- The problem of discovering “close matches” in the known function library.
- The problem of comparing behavioral specifications.
- Using supercomputers to attack the comparison problem, or divide the extraction.
- Applying pattern matching techniques to the simplification problem.
- ...