Abstract

SHARMA, TANU. Generation And Verification Of Software Robustness Properties Through Static Analysis (Under the direction of Jun Xu). Increasing reliance on computers calls for the need of robust software especially in critical applications such as those used in military, hospital etc. Traditional software testing techniques focus on functionality and ignore stressful conditions and exception handling. Poor programming practices may lead to critical software robustness failures resulting in memory corruption, application crashes and file system failures. Such robustness failures can be detected by many static analysis tools. However the difficulty in using existing tools is that they require users to provide robustness properties which need to be checked. Currently these properties which require source code and interface level information are mostly manually specified. This work proposes an FSA Generator framework that automatically generates concrete properties. Users only need to specify high level generic properties in simple finite state machines. The framework converts these generic properties into concrete language specific properties using source code information from a pattern database and interface level information from an API specification database. The automated cost effective generation of concrete properties makes static analysis scalable and efficient. Experimental evaluation using the generated properties and a static checker has found numerous robustness bugs in more than ten open source packages.
GENERATION AND VERIFICATION OF SOFTWARE ROBUSTNESS PROPERTIES THROUGH STATIC ANALYSIS

BY

Tanu Sharma

A THESIS SUBMITTED TO THE GRADUATE FACULTY OF
NORTH CAROLINA STATE UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
COMPUTER SCIENCE

RALEIGH
JANUARY 2006

APPROVED BY:

______________________________  ______________________________
David J. Thuente                Tao Xie

______________________________
Jun Xu

CHAIR OF ADVISORY COMMITTEE
To my parents.
Biography

Tanu Sharma was born in Meerut, UP, India. She received her Bachelors degree in Computer Science from the Institute of Engineering and Technology College, Lucknow, India. She has been a Masters student at the North Carolina State University, in the Department of Computer Science, since August 2003.
Acknowledgment

I would like to extend my foremost gratitude and appreciation to my advisor Dr. Jun Xu for his constant guidance and support throughout. Special thanks are also due to my co-advisor Dr. Thuente for giving me useful directions and motivating me every now and then. I am also grateful to Dr. Tao Xie for agreeing to be on my committee.

I would also like to thank my project partner and friend Mithun Acharya for adding basic data flow extentions to pushdown static analysis procedure used for verification. I also thank Qun He for building the initial version of the specification database used in the thesis. I am also thankful to my manager at Harris Corporation, RTP, Roger Boyer for his support and for providing me with the time I needed for the completion of this thesis.

This thesis would not have been possible without the unconditional support and trust my parents showed in me in sending me across thousand of miles away from home. Pursuing higher studies in a reputed university like NCSU was my childhood dream and I am thankful to my family for keeping it alive and helping me in making it real. I am indebted to my parents for giving me the values that help me today in taking pride in myself and be able to dream and achieve it. I thank my sisters, Anchal and Shefali, who have always helped me in moving ahead and are among my confidants and best friends. I am also thankful to my brothers-in-law, Madhur and Bhaskar jiju for their well-wishes.
I am thankful to all my friends who have always been a good support system for me. Special thanks to my best friend Nitu for giving me confidence and for being there always. I wish to again thank Mithun here for teaching me how to drive a car which made it possible for me to shuttle from work to college.

I am thankful to NC State University and USA for providing me with all the opportunities and facilities. Last but not least, I am highly thankful to all my teachers till now, who shaped me to who I am today and through this small acknowledgement I wish them to know how grateful I am to each one of them.
Table of Contents

List of Figures viii

1 Introduction 1
  1.1 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
  1.2 Roadmap for this Thesis . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

2 Motivation for FSA Generator 6
  2.1 MOPS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
  2.2 Using MOPS for checking Robustness Violations . . . . . . . . . . . . . . 8
  2.3 Representation of Properties . . . . . . . . . . . . . . . . . . . . . . . . . 9
    2.3.1 FSA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
    2.3.2 MFSA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
  2.4 Need for FSA Generator . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

3 FSA Generator Framework 13
  3.1 Generic robustness properties . . . . . . . . . . . . . . . . . . . . . . . . 15
  3.2 API Classification . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
  3.3 Language specific patterns . . . . . . . . . . . . . . . . . . . . . . . . . . 21
  3.4 Property Engine . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
  3.5 An Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
  3.6 Features and Advantages of FSA Generator . . . . . . . . . . . . . . . . . 28

4 Experiments and Results 29
  4.1 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
  4.2 API Robustness Violations . . . . . . . . . . . . . . . . . . . . . . . . . . 34
  4.3 Malloc Use Before Check Violations . . . . . . . . . . . . . . . . . . . . . 36

5 Conclusion and Future Work 38
  5.1 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2 Future Work</td>
<td>39</td>
</tr>
<tr>
<td>List of References</td>
<td>41</td>
</tr>
<tr>
<td>Appendices</td>
<td>44</td>
</tr>
<tr>
<td>A Errors Found in each package</td>
<td>45</td>
</tr>
<tr>
<td>B Malloc Use Before Check Error Code Snippets</td>
<td>48</td>
</tr>
</tbody>
</table>
# List of Figures

1.1 Code snippet for *vixie-cron* .................................................. 2  
2.1 FSA for *exec* ................................................................. 10  
2.2 MFSA for *execve* ............................................................... 12  
3.1 Property Engine ................................................................. 14  
3.2 The *Use Before Check* Generic Property ................................. 16  
3.3 API-DB ............................................................................. 18  
3.4 API-Classification ............................................................... 20  
3.5 *compareNULL* Pattern Example ............................................ 22  
3.6 Concrete *Use Before Check* property for *malloc* API call .......... 24  
3.7 *Use Before Check* Generic Property ..................................... 26  
3.8 FSA for *malloc* ................................................................. 27  
3.9 MFSA for *malloc* ............................................................... 27  
4.1 API calls selected for experiments ........................................... 31  
4.2 Packages .......................................................................... 33  
4.3 Robustness Errors ............................................................... 34  
4.4 Robustness Errors found for different API .................................. 35  
4.5 *malloc Use Before Check* Violations ..................................... 36  
4.6 *malloc Use Before Check* Violation in *dos2unix* ..................... 37  
4.7 *malloc Use Before Check* Violation in *routed* ....................... 37  
A.1 Violations found in packages tested ....................................... 46  
A.2 Violations found in packages tested cont’d ............................... 47  
B.1 Code snippets ................................................................. 49  
B.2 Code snippets cont’d .......................................................... 50  
B.3 Code snippets cont’d .......................................................... 51  
B.4 Code snippets cont’d .......................................................... 52
Chapter 1

Introduction

With the increasing importance of computers in present world, be it for small commercial use or for highly critical and dependable systems, the significance of software robustness is becoming more and more obvious.

How can we define software robustness and how can we measure it?

The IEEE Standard Glossary of Software Engineering Terminology (IEEE STD 610.12-1990)[1] defines robustness as the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environment conditions.

Where even general applications need a certain degree of robustness, the concern becomes unavoidable in cases of critical dependable systems such as telephone systems. Hospitals, banks, military organizations, etc. are using commercial software for highly secure and dependable applications. Correct functioning of these applications under exceptional or stressful settings is highly desired. Abnormal task termination or erroneous output may lead to severe loss of productivity or disruption of service.

To avoid such failures we need to understand what causes them. One of the reasons is stressful environment conditions, which may occur due to many factors such as exceptional input values, high computation load, memory exhaustion, process related failures and file
system failures. Many reliability failures occur due to incorrect use of system API calls. Programmer’s logical errors and lack of exception handling routines can lead to serious robustness failures. For example, dereferencing the return value of C memory management APIs on a NULL path leads to memory corruption leading to a system crash.

One such example can be seen in a redhat 9.0 package `vixie-cron`, where a robustness violation is caused due to dereference of a variable array without checking. Figure 1.1 shows the code snippet example for `vixie-cron` package. As can be seen in the example, the `malloc` API is used to allocate memory to the variable array `p`, which is being dereferenced and returned without any check. Usage without any check like this can lead to problems like memory corruptions and worse.

In small packages, there can be several programming errors like these, but in bigger packages, with more complex features and many more API calls, the possibility of presence of such weak points for robustness failures becomes more pronounced.

In this direction, there should be a method to directly find robustness violations in the source code. This can be achieved by static analysis which uses compilers to detect such bugs. For such an analysis, some robustness properties are defined which the application should conform to. Any violations reported while checking for these properties are robustness bugs that warn for possible system failures.

Such static checkers require properties to be represented as finite state machines. If
these finite state machines are to be written by hand, the whole purpose of using tools
to avoid manual effort to detect these errors in a code becomes insignificant. Moreover, a
manually written property can be erroneous. Also the process becomes monotonous and
tedious if the program is huge and needs to be checked for a lot of properties and API calls.

There are 282 POSIX API and one can imagine writing so many finite state machines
in case the need arises. This would even increase if the generic properties that need be
checked increase. The FSA Generator implemented in this thesis is able to do the task in
seconds and all that it needs as inputs are the API call and the generic property desired to be
checked. The automation engine then converts the generic property into the concrete ones
by using source code and interface level from the two databases AST-DB and API-DB.

Even if we carry out the manual effort once and write all finite state machines keeping
all the possible combinations of API calls and properties in mind, it might not be applicable
to all programming languages and leave out the properties that might become a concern in
future. The FSA Generator, is very extensible and the other generic properties can be
easily added to the framework as need may arise. Same is true for the pattern database and
modifications and extensions can be done quickly and easily. The FSA Generator is very
flexible and can be tuned to work with programming languages other than C.

The experiments done in this thesis found as many as 192 robustness violations in 12
open source packages. This enforces the significance of the FSA Generator in automated
static analysis.
1.1 Related Work

Research done in the direction of improving software robustness has resulted in various methodologies such as black box testing [4]. The black box method inserts stressful conditions directly or indirectly and studies the response of the application. Similar various other traditional software testing methods [19][22] focus on functionality of the application and thus its implementation correctness. These methods only make sure that the system gives expected output given a normal set of inputs. In a research project at University of Wisconsin, Fuzz [10][21][20], a program is subjected to random input streams. It studies the robustness of Unix system utilities and the results show that the failure rate of systems is quite high. Ballista [16][17], a Carnegie Mellon University research project, studied the robustness of different Unix operating systems while handling exceptional conditions. It uses extreme input parameter values to test the robustness of systems call interface implementation. Similar approaches have been applied by researchers at Reliable Software Technologies [12][23] and Bell Labs [29] for robustness testing of system library interfaces on Microsoft Windows. All these traditional methods do not address handling stressful environment conditions faced by software. Research [8] in this direction has proved that exception detection and handling code is the most ignored part of the software systems. These aspects are not understood completely and least tested in software testing. One such example is the field experience with telephone switch systems [27] that clearly shows that approximately two thirds of system failures are due to design faults in exception handling algorithms.

A follow up research in the Fuzz Project [21] tested how UNIX utilities checked return codes from the memory allocation routines by simulating the unavailability of virtual memory. Similar techniques at Reliable Software technologies were used to emulate exceptional

Some others [18][25][30] use static compiler analysis to find bugs and security holes in source code. The Meta-Compilation (MC) Project at Stanford University uses programmer written compiler extension to statically find bugs in operating systems [2][9] and cache protocols [7]. SLAM [3] and BLAST [13] are static analysis tools based on theorem proving and model checking Boolean abstractions of the program with iterative refinements. MOPS [6] is a static analysis checker that detects vulnerable system call sequences in the program.

1.2 Roadmap for this Thesis

The chapters in this thesis have been arranged as follows. Chapter 2 explains the static checker MOPS and representation of finite state automaton. This builds the understanding for the significance of the FSA Generator framework which is discussed in detail in Chapter 3. This chapter explains all the building blocks and the working mechanism of the framework. Chapter 4 evaluates the framework by discussing the experiments done with the framework for critical API calls. It includes relevant results and data for several packages selected for experimentation. This is followed by conclusion and discussion in Chapter 5.
Chapter 2

Motivation for FSA Generator

Ignorant and prudent programming practice can lead to software bugs in software where security is relevant. To avoid this, rules can be specified which governs the programming. These rules are known as temporal robustness/reliability/security properties. Once identified, these rules can be used to verify whether software code complies with such properties or not. Such verification can be manual for small pieces of code, but when scalability comes as an issue with the consideration of large programs, it becomes necessary to automate such a procedure. There have been several static tools available for detecting violations in a large and complex software code.

The focus of this thesis is auto-generation of properties used for static analysis. We use existing tools to enforce these properties in source code. One such static checker is MOPS which checks for security violations. We leverage it to detect robustness bugs.
2.1 MOPS

MOPS is a static analysis tool to automatically detect violations of a temporal security property in code. A temporal security property dictates the order of a sequence of security-relevant operations. Violation of this ordering constraint can lead to serious security attacks or even complete system compromise. For example, a temporal security property states that the call `strncpy(dst,src,n)` in a C program should be followed immediately by the statement `dst[n]='\0'`. Violation of such order makes it vulnerable to buffer overrun attacks.

MOPS ensures that properties are satisfied on all execution paths in the program. It makes use of model checking to determine this at compile time. Model checking techniques are utilized as we need to traverse each execution path.

It takes a program and security property, where security property is represented by a finite state automaton and the program is modeled as a pushdown automaton. All sequences of operations that violate the property end in the final (risky) states of the FSA. MOPS then determines whether certain states representing violation of the security property in the FSA are reachable in the PDA.

MOPS was initiated with the goals of soundness and scalability. Soundness is achieved through model checking of PDA. Scalability is mainly attained by MOPS by ignoring data flow analysis as it is quite expensive. MOPS is full inter-procedural, and thus it is efficient in finding inter-procedural bugs. Apart from these factors, another advantage of MOPS is modularization

Modularization is useful for practical use of the tool. Sometimes the security modes are very complex and it requires breaking up into simpler models which are easy to understand and describe. In the same way, as required, these simpler modules can be combined into complex ones.
There are several limitations in MOPS. Sometimes, there might be extra warnings (false positives) in the output as MOPS can mistakenly consider paths that are infeasible in the program to be feasible. Also, the stage when MOPS transforms a C program into a PDA is sound only as long as the program has no implementation-defined behavior.

2.2 Using MOPS for checking Robustness Violations

The original MOPS checks for temporal safety property violations, that means checking whether the program performs certain operations in a specific order or not. For checking robustness violations, we need the same kind of checking to be performed where we define a set of sequence order of certain operations and check whether that sequence is followed or not. But original MOPS is not suitable to be used to check for robustness violations, as it is data flow insensitive. Robustness property is defined such that it requires tracking a variable and checking for its value before the function is returned. Because of the data flow insensitivity, MOPS ignores data values and assumes that a variable may take any value. As we know that lack of exception handling routines lead to robustness failures, static analysis tracks such procedures. Exception handling procedures are usually characterized by conditional constructs that check the return value of an API call. But the original MOPS assumes that both branches of a conditional statement may be taken. Our experiments use a modified MOPS which tracks the values of variables that take the return value of an API call along the different branches of conditional constructs. For each possible execution sequence, MOPS associates a value to the variable that is being tracked. Since MOPS is path specific, the correct binding of the variable to a value is retained along all execution sequences. This tracking of variables is done by pattern matching supported by MOPS. This pattern matching provided syntactic meaning which is language specific. A robustness
failure is inconsistency reported in the tracked variable.

The modified MOPS is intra-procedural. Future work may make it inter-procedural such that a variable can be tracked in different procedures.

2.3 Representation of Properties

MOPS requires each property in the form of a finite state automata.

2.3.1 FSA

A FSA is described in a text file where each statement ends with a semicolon (‘;’). The three types of statements in the FSA file can be identified with the following keywords:

- **state**: This statement declares a state. For example:
  
  ```
  state start “state start”;
  ```
  
  A finite state automaton consists of several states such as a start state and an end state. All sequences of operations that result in the violation of the property end in the final risky states of the FSA. In the above example, a state named `start` is defined and labeled as “state start”.

- **ast**: This statement declares an Abstract Syntax Tree (AST) that MOPS needs to check while looking through the source code for any violation. For example:
  
  ```
  ast execvePlain { function_call { identifier “execve”} {ellipsis}};
  ```
  
  This statement matches `execve()` in a source code. Patterns are discussed more in detail in Chapter 4.

- **transition**: This statement declares a transition from one state to another. For example:
  
  ```
  transition start execveed execvePlain;
  ```
This statement defines a transition from state `execveed` to another state named `execvePlain`.

Figure 2.1 shows an example of a FSA for API call `execve` that is needed by MOPS to check for a property in a given code. Without going into details, the basic structure of FSA can be seen in its three parts.
2.3.2 MFSA

There is another small file required by MOPS to locate the right FSA and perform checking. It is called meta-FSA or simply MFSA. A complex property may be decomposed into a set of simpler ones described by FSAs. MFSA represents a FSA that is the product of multiple FSAs. It is again a text file with statements ending with semicolon.

It consists of statements starting with the following keywords:

• **label**: This statement describes a meta-FSA and is optional. Each MFSA can have only one label statement. For example:

  ```
  label "Checking for violation";
  ```

• **fsa**: This statement consists of all the FSAs that form this MFSA. It may also consists of only one FSA. For example:

  ```
  fsa "execvCheckShouldExist.fsa";
  ```

• **initial state**: This statement declares an initial state of the meta-FSA and its description. It may consist of one or more initial states depending on the number of FSAs that are included in the meta-FSA. For example:

  ```
  initialstate start "start";
  ```

• **final state**: This statement declares a final state of the meta-FSA followed by its description. It may consists of one or more final states depending on the number of FSAs that are included in the meta-FSA. For example:

  ```
  finalstate checkShouldExistError "Error: checkShouldExist";
  ```

Figure 2.2 shows the meta-FSA for the `execve` API call.
MOPS is a tool for static analysis which checks for violations for security properties. These properties are temporal in nature and require a specific order in which operations are expected to be performed. In this thesis we aim towards finding robustness violations making use of the model checking techniques provided by MOPS. Thus we define robustness rules for safe programming practice and pass them as inputs into modified MOPS for checking.

As MOPS needs FSA and MFSA as inputs to perform model checking, the user has to generate a finite state machine manually. The manual effort carried in this direction can be estimated based on the following data: there are 282 API calls in the POSIX standard and there is a large number of properties for which the code needs to be checked for robustness bugs. The process of manually writing language and interface properties that can be used by a static checker is time-consuming and requires that the user should have the knowledge and skill to design and write such FSAs that may become quite complicated for some properties. If one wants to perform a check for several API calls and a number of property violations in code, the checking seems infeasible and is not cost-effective. Thus in order to utilize MOPS to detect violations, it becomes imperative to automate the procedure of generating finite state machines. The FSA Generator framework is developed with such an objective in mind. The following chapter discusses the framework in detail and enforces its significance in static analysis for detecting robustness failures.
Chapter 3

FSA Generator Framework

The static checker MOPS requires concrete properties with source code and interface level information in order to detect the violations in the code. These concrete properties are represented by finite state machines. Writing large number of such concrete properties is very labor intensive and can be erroneous. The automatic checking advantages provided by MOPS are over-shadowed by the human cost involved in creating its input. Moreover, the users using MOPS for static checking need to have knowledge of all the information that form a concrete property. Thus the task of writing concrete properties requires a lot of human effort and knowledge, and is time-consuming. Thus there is a need for automating such a procedure of generating concrete properties. The framework for this task should be able to keep the details hidden from the user and the relevant source code, interface level and language specific information should be taken into consideration. The FSA Generator framework provides such an automation for generating concrete properties. MOPS takes these concrete properties and program code as inputs and checks whether there is a violation of these properties.

Before understanding the full mechanism of the automation engine, it is important to understand what makes the FSA Generator Framework and how its various components
interact. The entire framework has the following components:

- Generic properties database
- API-DB
- AST-DB
- Property or Automation Engine

Figure 3.1 depicts the interaction among these components of the framework. Properties in the generic properties database have no concrete information. The automation engine translates a generic robustness property into a concrete property using source code level information from a programming language specific pattern database (AST-DB) and interface level information from an API specification database (API-DB). Thus these two databases provide the needed inputs to the framework in order to carry out the translation.

This chapter explains all the building blocks of the framework. This is followed by a detailed discussion of its working mechanism. It also covers the advantages of using such a framework for detecting robustness failures.
3.1 Generic robustness properties

A generic property for an interface $i$ can be defined as follows. Assume the following for the execution of an interface $i$:

- $rs(i)$: Result set of $i$ consisting of variables containing the result of $i$’s execution.
- $ss(i)$: Status set of $i$ consisting of variables indicating success or failure status.

A simple property *Use Before Check* states: *For an interface $i$, an element in $rs(i)$ can be used only after an element in $ss(i)$ is checked for error on all execution paths.*

In the FSA Generator framework, these properties are represented by simple finite state machines. Figure 3.2 shows a simple finite state machine for *Use Before Check* property. As can be seen, the generic property makes use of abstract and generic terms. In this finite state machine, $api$ is the transition from the initial $start$ state to the $called$ state. In this finite state machine, $error$ state represents the robustness violation and the $checked$ state is the safe state. Thus if the result of an API call is used before its error status is checked, the end state is the error state and the violation is reported. For example, consider API Call *malloc* which returns NULL on failures, thus the *Use Before Check* property for *malloc* states that the return value of *malloc* should be checked for NULL before it can be used. If no such check is made and the return value of *malloc* is used, it is a violation to the property.

In the current implementation, the framework defines the following generic properties which cover some of the most common robustness violations, however, the framework is extensible and new generic properties can be easily defined by a user using a simple finite state automata.

- *Use Before Check*: This generic property applies to the API calls that return a NULL pointer (the class of such API calls is termed as [p:np]). It checks for the violations that occur because of using a variable that holds return value of
such an API before checking.

- **Check Should Exist**: All those API calls that return an integer value on success or failure, should be checked before use. For example `closedir` returns ‘0’ on success and ‘-1’ on failure.

- **Null Pointer Dereferencing**: For API calls in p:np class, a variable holding the return value should not be dereferenced on a NULL path.

- **Null Pointer Free**: For API calls in p:np class, a variable holding the return value should not be freed on a NULL path.

- **Free Pointer Dereferencing**: For API calls in p:np class, a pointer variable that is freed once should not be dereferenced along all execution paths.

- **Double Free**: For API calls in p:np class, a pointer variable that is freed once should not be freed again along all execution paths.

Currently there are the above six properties which are quite common but new properties can be easily added to the framework. The generic properties database consists of a separate property file for each property. To keep these property files generic, the naming convention has kept to be abstract and these files are extensible. Whenever the framework needs to be
extended to support a new property, a new file can be written and simply included in the system. It requires minimal changes in the engine to make it recognize and handle the new property.

These files consist of the generic states and transitions which are checked for a property. The FSA Generator generates a specific concrete property output making use of these generic files.

For example, for property Use Before Check, the generic property file consist of the following states:

- state start “state start”;
- state API called “state API called”;
- state useBeforeCheckError “state useBeforeCheckError”;
- state checked state “checked”;

For malloc API call and Use Before Check property, the output finite state automaton would consist of the following specific states:

- state start “state start”;
- state mallocCalled “state mallocCalled”;
- state useBeforeCheckError “state useBeforeCheckError”;
- state NULLCompared “state NULLCompared”;
- state notNULLCompared “state notNULLCompared”;

### 3.2 API Classification

In the process of translating generic properties into concrete ones, the interface level information comes from the API database. This database includes all the 282 POSIX APIs
Figure 3.3: API-DB

functions. **POSIX** stands for Portable Operating System Interface. The entries in the database show the input parameters for the API call, return value types, return values on success/failure and the error flags set.

The database consists of the API call information in the following record format:

\[ [\text{sequential number}]:[\text{function name}]:[\text{parameter list}]:[\text{return type}]:[\text{return value on success}]:[\text{return value on failure}]:[\text{error number list}] \]

Figure 3.3 shows several important specifications.

The FSA Generator queries the database for a specific API, and obtains its interface level information in the form of its return values on success/failure and error specifications. This knowledge is used in order to convert the generic properties into concrete ones.

According to the return values on success and failure, the APIs are classified into different classes. Figure 3.4 shows the classification. It also includes an example API member for each class. After identifying a particular class for an API, the FSA Generator picks up the right source code level information from a pattern database.

A class in API classification is represented as:

\[ [\text{Return Values On Success}: \text{Return Value on Failure}] \]

For example, members in API class [0:-1] return ‘0’ on success and ‘-1’ on failure. Those in [x,0:EOF] returns some value ‘x’ or ‘0’ on success and ‘EOF’ on failure. In the

<table>
<thead>
<tr>
<th>API</th>
<th>parameter list</th>
<th>return type</th>
<th>return value on success</th>
<th>return value on failure</th>
<th>errno</th>
</tr>
</thead>
<tbody>
<tr>
<td>chmod</td>
<td>const char * path , ...</td>
<td>int</td>
<td>0</td>
<td>-1</td>
<td>EPERM, ...</td>
</tr>
<tr>
<td>open</td>
<td>const char * pathname, ...</td>
<td>int</td>
<td>fd</td>
<td>-1</td>
<td>EEXIST, ...</td>
</tr>
<tr>
<td>malloc</td>
<td>size_t size</td>
<td>void*</td>
<td>pointer</td>
<td>null pointer</td>
<td></td>
</tr>
<tr>
<td>fseek</td>
<td>FILE * stream , ...</td>
<td>int</td>
<td>0</td>
<td>-1</td>
<td>EBADF, ...</td>
</tr>
<tr>
<td>remove</td>
<td>const char * pathname</td>
<td>int</td>
<td>0</td>
<td>-1</td>
<td>EFAULT, ...</td>
</tr>
</tbody>
</table>
classification, ‘x’ refers to some undefined value, ‘p’ is the returned pointer for API calls such as `malloc` etc., ‘np’ stands for null pointer, ‘EOF’ is end of file and ‘negative’ stands for ‘<0’ values. There are some classes which return no value on success or failure. An example of such a set of classes is [1,0:]. Classes which do not return anything on success or return, such as [:], are those that do not require any checks and thus the supported properties do not apply to them, such as [x:x] or [p:p], are not included in the classification.

For any API class, the properties are defined in a manner that requires checks for the values returned. For example, for [p:np] class, the code should have ‘NULL’ value checks before the pointer variable can be used. This can be done in several ways like whether the variable is equal to ‘NULL’ or it is not equal to ‘NULL’ etc. Similarly for class say [0:-1], the checking looks for patterns like ‘==0’, ‘!=0’, ‘==1’, ‘!=1’ in the code.

Such type of pattern matching groups several subclasses together as can be seen in the classification. Thus for classes [-1,1,0:-1], whose members return ‘-1’ or ‘1’ or ‘0’ on success and ‘-1’ on failure and [1,0:-1], where return values on success are ‘1’ or ‘0’ and ‘-1’ on failure, same comparison patterns are searched for, that is, ‘==’ and ‘!’ with ‘1’, ‘-1’ and ‘0’. All the classes are termed subclasses, and those which require the same patterns are grouped under an API class which is named after the simplest among all its subclasses.

Identifying a class is also important as several properties should not be checked for some API classes. These properties are applicable to the API calls on the basis of their return values. For example, those which do not have “null ptr” as their return value cannot have any of the properties which apply to pointers. Examples are Double Free, Null Pointer Dereference or Use Before Check.

Likewise, those which do not have a constant or numeric return value, cannot be checked for property Check Should Exist. For example, the [p:np] class API can be checked for the following properties:
<table>
<thead>
<tr>
<th>No.</th>
<th>API class</th>
<th>sub classes</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>0:-1</td>
<td></td>
<td>closedir</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x,0:-1</td>
<td>mblen</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fd,flag,fld_owner,signal,0:-1</td>
<td>fcntl</td>
</tr>
<tr>
<td>II</td>
<td>0,1,-1:-1</td>
<td>-1,1,0:-1</td>
<td>sysconf</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1,0:-1</td>
<td>readdir</td>
</tr>
<tr>
<td>III</td>
<td>0:EOF</td>
<td>0:EOF</td>
<td>fflush</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x,0:EOF</td>
<td>scanf</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0:EOF,undefined</td>
<td>fclose</td>
</tr>
<tr>
<td>IV</td>
<td>0:x</td>
<td>0:x</td>
<td>ttyname_r</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x,0:0</td>
<td>strftime</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x,0::</td>
<td>isupper</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0:0</td>
<td>sem_wait</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0:0:0</td>
<td>setbuf</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0:0::</td>
<td>setjmp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0,0::</td>
<td>alarm</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0:0::</td>
<td>setjmp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0:0:0::</td>
<td>setjmp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x,p,0:-x,0</td>
<td>strtod</td>
</tr>
<tr>
<td>V</td>
<td>1,0::</td>
<td>1,0::</td>
<td>isatty</td>
</tr>
<tr>
<td>VI</td>
<td>np,-1,0;x np,-1,0;x</td>
<td>np,-1,0;x np,-1,0;x</td>
<td>wctomb</td>
</tr>
<tr>
<td>VII</td>
<td>p:np</td>
<td>p:np</td>
<td>malloc</td>
</tr>
<tr>
<td></td>
<td></td>
<td>p:pp:np</td>
<td>striok</td>
</tr>
<tr>
<td></td>
<td></td>
<td>p:pp:pp:np</td>
<td>realloc</td>
</tr>
<tr>
<td>VIII</td>
<td>x;-1</td>
<td>x;-1</td>
<td>ftell</td>
</tr>
<tr>
<td></td>
<td></td>
<td>::;-1</td>
<td>feof</td>
</tr>
<tr>
<td></td>
<td></td>
<td>noreturn;-1</td>
<td>exeve</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x,-1:-1</td>
<td>pathconf</td>
</tr>
<tr>
<td>IX</td>
<td>x:EOF</td>
<td>x:EOF</td>
<td>putchar</td>
</tr>
<tr>
<td>X</td>
<td>x,EOF:np</td>
<td>x,EOF:np</td>
<td>getchar</td>
</tr>
<tr>
<td>XI</td>
<td>x:negative</td>
<td>x:negative</td>
<td>printf</td>
</tr>
</tbody>
</table>

**Figure 3.4:** API-Classification
• Use Before Check
• Null Pointer Dereferencing
• Null Pointer Free
• Free Pointer Dereferencing
• Double Free

Rest of the other classes can be checked for Check Should Exist property.

3.3 Language specific patterns

The patterns database includes the syntax and semantics of the API calls and the various components of exception handling code like checking routines and variables usages in Abstract Syntax Tree (AST) notations. The transitions in the generic finite state machine have no language level semantic. Thus the FSA Generator converts these abstract transitions into the programming language specific transitions to be included in the final output FSM by using information provided in the patterns database termed here as AST-DB. The FSA Generator queries the patterns database using the return values on success/failure and error status and the database gives the right patterns containing the right source code level information.

For example, for the property Use Before Check, the malloc API call returns a memory pointer on success and a NULL pointer on failure. Assuming pointer variable contains the return value of the malloc call, the check status implies that p is compared to NULL. This can be done in various ways such as (p == NULL) , (!p), (p) , (p !=NULL) etc. The pattern database returns all such possible patterns for checking of a NULL pointer.

An example of such a pattern for comparison with NULL can be seen in Figure 3.5. It depicts an example of compareNULL ast pattern , which compares to value NULL in
various possible ways. Consider checking robustness violation for API call malloc. The ast pattern compares the return value to NULL in various ways. For example, consider the expression in the pattern

\[
eq \ "\ " \ \{ \ var \ x \} \ { \ cast \ \{ \ lexical \_ cst \ 0 \} \} \\
\]  

The statement can be translated to this: \((p==\text{NULL})\) where \(p\) is the pointer variable holding the return value of malloc API call represented by ‘\(x\)’. In the above expression, \(\text{cast}\{\text{lexical}\_\text{cst} \ 0\}\) compares to NULL value.

The next statement:

\[
eq \ "\ " \ \{ \ assign \ "\ " \ \{ \ var \ x \} \ \{ \ cast \ \{ \ function\_call \ \{ \ identifier \ "\ident\_str\" \} \{ \ ellipsis \} \} \ \{ \ cast \ \{ \ lexical\_\text{cst} \ 0\} \} \} \}} \\
\]

This can be translated to something like this for example:

\((p=(\text{int} \ *)\text{malloc}(\text{sizeof(int)))==\text{NULL})\)

In this expression, \(\text{var} \ x\) is the variable which is the return value of malloc API. In this case, \(\text{cast}\) matches the cast of the function malloc which is (\text{int} \ *) here. In this case, \(\text{ellipsis}\) matches zero or more ASTs. Thus the following matches \(\text{printf} (...)\) for example:

\[
\{ \text{function\_call} \ \{ \ \text{identifier} \ \text{printf} \} \ \{ \ \text{ellipsis} \} \} \\
\]

The example consists of other statements that consist of expressions matching to function without cast and matching other possible ways of comparing to NULL value.
3.4 Property Engine

As discussed previously, the property engine takes inputs from API-DB and AST-DB and generates FSA and MFSA files. It converts abstract terms from these databases into specific to those for a given API call and thus results in concrete properties. These properties are specific to an API call and language for which patterns are supported in the framework. This can be understood by studying the generated concrete property of the previously discussed example of *Use Before Check* property for malloc API call. In the concrete FSM, each directed edge and vertex from the generic FSM could be split into multiple edges and vertices.

Figure 3.6 shows the generated concrete property FSM for the malloc API Call from the previously considered generic property. As we can see, the *check* keyword is split into different edges labeled by the NULL pointer check patterns and the *use* keyword is replaced by patterns denoting different ways in which a pointer variable can be used. Figure 3.6 discusses the case *p=malloc()*.* If ‘p’ is checked for ‘NULL’ value, which can be done in various ways, for example, ‘p!=NULL’, ‘p==NULL’, ‘!p’ etc, the execution ends in safe state. If ‘p’ is used in any way without any check for example, ‘*p’, ‘p[]’, ‘p->x’ etc., a robustness violation is reported.

The FSA Generator currently supports 282 API calls and six properties. The engine is intelligent enough to ignore the API calls with no return values and no error codes. Also, there are several API calls for which no checking is needed. Some of these API calls have the same return values on success and failure. For example, values returned are x:x or p:p or -x:-x or -x: : etc. An example of such an API is fread.

Also, not all the properties can be checked for each API call. As explained in the section of API classification, all the [p:np] class members can be checked for five properties and
the rest others can be checked for the Check Should Exist property. There might also exist a class to which all the properties may apply, for example, a class \([x,-1,np,0:x,-1,np,0]\). The engine inspects all such possible cases and in case of an expected result, it generates the output FSA file or do nothing.

The FSA Generator is very simple to use and gives output files in seconds. One or more finite state machines can be generated at one time. Output files can be generated with any of the following options on the command line:

- `java fsaGenerator apiCall property`
  Gives FSA and MFSA files for the specific property and API call.

- `java fsaGenerator apiCall all`
  Gives FSA and MFSA files for the specific API call against all the properties.

- `java fsaGenerator all property`
  Gives FSA and MFSA files for all API calls against the specific property.

- `java fsaGenerator all all`
  Gives FSA and MFSA files for all possible combination of API calls and properties.

![Figure 3.6: Concrete Use Before Check property for malloc API call](image)
“apiCall” is the API call for which we need the FSA and MFSA files. “property” is the name of the property, which is being checked.

Thus we see that the user need not have to be aware of all the details about the finite state automaton or worry about writing it. Just by providing the API call and the generic property desired to the FSA Generator, the concrete property can be obtained.

The next section explains a simple example that knits together all the sections discussed so far.

### 3.5 An Example

Here is an example of how the FSA Generator generates FSA output files. In this section, we assume that we want to check for a robustness violation for property *Use Before Check* for API call *malloc* in the given code.

In order to check for a robustness violation for an API call, the following command is used:

```java
java fsaGenerator malloc useBeforeCheck
```

Using this command, the FSA Generator creates a concrete property FSA describing *Use Before Check* generic robustness property that states: “Check the returned value of malloc before use”.

Knowing which property to check, the FSA Generator goes through the properties database and picks up the file containing the generic property defined for *Use Before Check*. Figure 3.7 depicts the generic property file for *Use Before Check* that consists of states and transitions. This can be graphically represented by Figure 3.2.

As soon as the FSA Generator knows which API to use to make FSA, it picks up the interface level information from the API database. For *malloc*, the retrieved information is
as follows:

2:\texttt{malloc:}:\texttt{size\_t}:\texttt{size}:\texttt{void}:p:np:

Now the FSA Generator knows that the \texttt{malloc} API call returns ‘p’ on success and ‘np’ on failure. Based on this knowledge, the FSA Generator identifies the API class \texttt{malloc} belongs to, which is \texttt{[p:np]} in this case. Once this is done, the automation engine knows which patterns to pick up from the pattern database AST-DB. AST patterns consist of the source code level information needed for the finite state automaton. As \texttt{malloc} belongs to \texttt{[p:np]} class and the property is \textit{Use Before Check}, the check could be present in the form ‘\texttt{==NULL}’ or ‘\texttt{!=NULL}’. Thus the correct ‘\texttt{compareNULL}’ and ‘\texttt{compareNotNULL}’ ast patterns are picked up along with the other usual ones.

Figure 3.8 shows the generated FSA file and Figure 3.9 depicts the required MFSA file. As the \textit{Use Before Check} property is a simple property and not a complex one, the meta-FSA consists of only one FSA. Graphically the FSA can be represented as in Figure 3.6.
Figure 3.8: FSA for malloc

Figure 3.9: MFSA for malloc
3.6 Features and Advantages of FSA Generator

The key advantages of the FSA Generator are its simplicity and usefulness in static analysis checking. It can be tuned to support more languages and facilitates adding new functionalities to the system. All this can be achieved through its several features:

- **Automation**: The FSA Generator automatically generates as many concrete finite state machines as needed by taking inputs from the previously defined generic properties and the databases AST-DB and API-DB. The automation saves manual effort and time which otherwise might have made property generation non-scalable and error-prone.

- **Transparency**: The FSA Generator makes use of source code and interface level information from the AST and API databases while the user is completely unaware of this. The user just has to deal with generic properties that are defined as simple finite state automata. The details are all hidden from whoever uses it for generating concrete properties.

- **Extensible**: The framework can support as many API calls by simply expanding the API database. To add a property, a simple generic property needs to be written which mostly consists of very simple recognizable terms. In addition, more languages can be supported. Currently patterns for only C are supported but the AST database can be extended to include patterns for other languages. Only small changes need to be done within the framework to make it work with the newly added patterns and properties.
Chapter 4

Experiments and Results

In order to evaluate the practical use and significance of the framework, commonly used open source packages were tested. Most of these are from Redhat 9.0 distribution. The configuration used was a Pentium IV machine with 2.8 GHz processor speed and 1GB RAM running on Fedora Code 32.6.9-1.667smp kernel. The framework generates concrete properties from the 6 generic properties we currently have.

Due to experimentation resource constraints (mainly long run time), out of the 282 POSIX API, 60 critical API calls were selected. These are frequently used API calls in common and critical operations such as memory management, file and string I/O and permission management.

Robustness failures can be classified as critical or minor. This can be done on evaluating recovery costs in case of such failures. In general, critical failures include system and application crashes and minor failures include hindering cases. Ballista [19] suggests a failure mode classification called CRASH. It is an acronym for the following failure modes:

- **Catastrophic Failures**: These failures result in a complete system crash and requires a system reboot.
- **Restart Failures**: In such cases, the application hungs and requires application
restart.

- **Abort Failures**: The application is abnormally terminated in this case.
- **Silent Failures**: They occur when the called function or system call is completed successfully instead of returning an error indication as expected due to the invalid parameter values used in the call.
- **Hindering Failures**: These result in incorrect error indication such as the wrong error reporting code.

In the comparison of robustness of POSIX operating systems [15], several examples of catastrophic failures due to `malloc` API call are discussed. It also lists data types which are most commonly associated with abort robustness failures. The results show that NULL and invalid pointer values are the most common causes of abort failures. We also see that string and math functions are also responsible for significant number of robustness failures. It lists several system calls as having high failure rates.

Selection of API Calls for experiments in this thesis has been made on the basis of their frequent use in applications. In case of improper usage of these calls, critical failures may happen. These API Calls are mainly which are used for memory operations for example, `malloc, realloc, calloc`, permission management such as `chmod, setuid`, file and directory operations for example, `closedir, fopen, fileno` and string operations for example, `strrchr`.

The selected API calls are listed in Figure 4.1. Some of these are applicable to be checked for **Use Before Check** property and the rest for **Check Should Exist** based on their return values as discussed in Chapter 3. Thus they can be divided into two groups as follows:

- **Check Should Exist**: `access, chdir, chmod, chown, close, closedir, creat, execv, execve, fchdir, chmod, fchown, fclose, feof, fflush, fgetc, fget, fileno, fputs, fputc, fseek, fstat, ftell, getc, getchar, lchown, link, lseek, open, putc, putchar, puts, ...`
Figure 4.1: API calls selected for experiments

readdir, rename, setgid, setuid, stat, ungetc, unlink, wait, write

- Use Before Check: calloc, fdopen, fgets, fopen, freopen, getcwd, getenv, getgrgid, getgrnam, getpwnam, getpwuid, gets, getwd, malloc, opendir, strrchr, strstr, tmpnam, ttyname

For the experiments, 12 widely used open source packages were used. Figure 4.2 lists these packages. Each package is briefly discussed here:

- ftp-0.17-17: The ftp package provides the standard UNIX command-line FTP (File Transfer Protocol) client. FTP is a widely used protocol for transferring files over the Internet and for archiving files.

- ncompress-4.2.4-33: The ncompress package contains the compress and uncompress file compression and decompression utilities, which are compatible with the original UNIX compress utility (.Z file extensions).

- routed-0.17-14: It is the RIP-based routing daemon used by most UNIX systems to handle the routing tables. It can exchange RIP messages with other machines, updating its route tables as necessary.

- rsh-0.17-14: The rsh package contains a set of programs which allow users to access remote machines, run commands on remote machines, login to other machines and copy files between machines (rsh, rlogin and rcp).
• **sysklogd-1.3.31-3**: The *sysklogd* package contains two system utilities (syslogd and klogd) which provide support for system logging. Syslogd and klogd run as daemons (background processes) and log system messages to different places, like sendmail logs, security logs, error logs, etc.

• **sysstat-4.0.7-3**: This package provides the sar and iostat commands for Linux. Sar and iostat enable system monitoring of disk, network, and other IO activity.

• **SysVinit-2.84-13**: The *SysVinit* package contains a group of processes that control the very basic functions of the system. *SysVinit* includes the init program, the first program started by the Linux kernel when the system boots. Init then controls the startup, running, and shutdown of all other programs.

• **tftp-0.32-4**: The Trivial File Transfer Protocol (TFTP) is normally used only for booting diskless workstations. The *tftp* package provides the user interface for TFTP, which allows users to transfer files to and from a remote machine.

• **traceroute-1.4a12-9**: The *traceroute* utility displays the route used by IP packets on their way to a specified network (or Internet) host. *traceroute* displays the IP number and host name (if possible) of the machines along the route taken by the packets. *traceroute* is used as a network debugging tool.

• **zlib-1.1.3-3**: The *zlib* compression library provides in-memory compression and decompression functions, including integrity checks of the uncompressed data. This library is used by a number of different system programs.

• **ltrace-0.3.29-1**: *ltrace* is a debugging program which runs a specified command until the command exits. While the command is executing, *ltrace* intercepts and records both the dynamic library calls called by the executed process and the signals received by the executed process. *ltrace* can also intercept and print system calls executed by the process.
• pciutils-2.1.10-7: The PCI Utilities package contains a library for portable access to PCI bus configuration registers and several utilities based on this library. The utilities include lspci which displays detailed information about all PCI busses and devices and setpci which allows to read from and write to PCI device configuration registers.

### 4.1 Evaluation

Concisely, the total number of robustness violations found in each package can be seen in Figure 4.3(a). As can be seen in the table, these commonly used open source packages have unavoidable number of robustness violations. These violations include several critical APIs which can lead to critical robustness failures. Only 12 of these packages showed 192 violations for just two properties.

Figure 4.3(b) lists the violations in one of these selected packages. As can be seen, a critical package SysVinit, which contains a group of processes that control the very basic functions of a system, reported as many as 64 robustness violations. In the table we can see several unavoidable robustness violations such as that of fclose, malloc. Such violations
Figure 4.3: Robustness Errors can lead to failures such as memory corruption, file system failures etc. Thus it is important to detect such failures and correct them.

It is important to see how many critical API violations are present in these experiments. Errors found in each package can be seen in their individual tables in Appendix A.

### 4.2 API Robustness Violations

In the experiments done for 60 API Calls on the 12 packages, 22 API calls showed robustness violations. Figure 4.4 shows the total number of violations found in the packages for each API call. The table shows that the violations consist of several critical API calls corresponding to operations such as that of memory management, file system, string handling etc. Some API calls which are very commonly used such as `close`, `fclose` etc, shows a surprising number of violations and if left unchecked, these violations can lead to corruption to file system etc. The table shows that 8 `malloc Use Before Check` errors were detected.
<table>
<thead>
<tr>
<th>API call</th>
<th># errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>close</td>
<td>42</td>
</tr>
<tr>
<td>closedir</td>
<td>4</td>
</tr>
<tr>
<td>chdir</td>
<td>3</td>
</tr>
<tr>
<td>fseek</td>
<td>4</td>
</tr>
<tr>
<td>fdopen</td>
<td>3</td>
</tr>
<tr>
<td>fgets</td>
<td>1</td>
</tr>
<tr>
<td>fclose</td>
<td>47</td>
</tr>
<tr>
<td>fflush</td>
<td>23</td>
</tr>
<tr>
<td>fputc</td>
<td>2</td>
</tr>
<tr>
<td>fputs</td>
<td>6</td>
</tr>
<tr>
<td>fileno</td>
<td>8</td>
</tr>
<tr>
<td>ftell</td>
<td>1</td>
</tr>
<tr>
<td>fstat</td>
<td>4</td>
</tr>
<tr>
<td>getenv</td>
<td>1</td>
</tr>
<tr>
<td>getpwuid</td>
<td>1</td>
</tr>
<tr>
<td>malloc</td>
<td>8</td>
</tr>
<tr>
<td>open</td>
<td>2</td>
</tr>
<tr>
<td>putchar</td>
<td>8</td>
</tr>
<tr>
<td>puts</td>
<td>2</td>
</tr>
<tr>
<td>setuid</td>
<td>5</td>
</tr>
<tr>
<td>unlink</td>
<td>7</td>
</tr>
<tr>
<td>write</td>
<td>9</td>
</tr>
</tbody>
</table>

**Figure 4.4:** Robustness Errors found for different API
4.3 Malloc Use Before Check Violations

This section covers all the violations for malloc API call found in the experiments done. Improper use of certain critical API calls such as malloc may result in catastrophic failures. For this purpose, several more experiments were conducted. In addition to the packages used above, other two important open source packages were:

- **dos2unix-3.1-15**: It converts DOS or MAC text files to UNIX format.
- **vixie-cron-3.0.1-74**: The vixie-cron package contains the Vixie version of cron. Cron is a standard UNIX daemon that runs specified programs at scheduled times. Package vixie-cron adds better security and more powerful configuration options to the standard version of cron.

In total, 11 malloc Use Before Check violations were found in these 14 packages which can lead to system failure. They are listed in Figure 4.5.

Figure 4.6 shows one such violation in an open source package dos2unix. In this case, pFlag is used without being checked for NULL value. Another malloc violation can be seen in open source package routed as shown if Figure 4.7.

Code snippets for all the malloc Use Before Check errors found can be found in Appendix B.

<table>
<thead>
<tr>
<th>packages</th>
<th># malloc errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>neomdoc-4.24-33.src.rpm</td>
<td>1</td>
</tr>
<tr>
<td>dos2unix-3.1-15.src.rpm</td>
<td>1</td>
</tr>
<tr>
<td>routed-0.17-14.src.rpm</td>
<td>1</td>
</tr>
<tr>
<td>fip-0.17-17.src.rpm</td>
<td>3</td>
</tr>
<tr>
<td>iftrace-0.3.29-1.src.rpm</td>
<td>1</td>
</tr>
<tr>
<td>SysVinit-2.84-13.src.rpm</td>
<td>1</td>
</tr>
<tr>
<td>vixie-cron-3.0.1-74.src.rpm</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 4.5: malloc Use Before Check Violations
Figure 4.6: malloc Use Before Check Violation in `dos2unix`

```c
ArgIdx = 0;
CanSwitchFileMode = 1;
ShouldExit = 0;

pFlag = (CFlag*)malloc(sizeof(CFlag));
pFlag->NewFile = 0;
pFlag->Quiet = 0;
pFlag->KeepDate = 0;
pFlag->ConvMode = 0;
pFlag->NewLine = 0;
```

Figure 4.7: malloc Use Before Check Violation in `routed`

```c
externalinterfaces++;

ifp = (struct interface *)malloc(sizeof (*ifp));
memset(ifp, 0, sizeof (*ifp));
ifp->int_flags = IFF_REMOTE;
/* can't identify broadcast capability */
ifp->int_net = inet_netof_subnet(dst.sin_addr);
```
Chapter 5

Conclusion and Future Work

This chapter summarizes the thesis and discusses directions for future work.

5.1 Conclusion

Robustness failures are becoming an increasing concern with increase in use of commercial software in critical applications. Thus it becomes important to provide more reliable and cost effective software validation to such applications. Among several present techniques and tools available for automated testing of robustness bugs in a software, this thesis considers a modified MOPS for experimentation purposes. MOPS requires intelligent concrete properties which are automatically generated by the framework presented in this study.

The FSA Generator framework implemented and presented in this thesis is very easy to use. The core of the framework is the automation engine which translates the simple abstract properties into concrete properties. These concrete properties are language specific because of the inputs from the AST-DB of the framework and carry interface level information picked up from API-DB. The users deal with the generic property and thus these code level details are hidden from them. The AST-DB consists of patterns that specify the
language-level transitions in the concrete property. The API-DB lists all the API calls and the information which is required by the engine to decide whether a concrete property is possible for that API call and if it does, generates it. This database carries information for 282 POSIX API calls and the AST-DB consists of all the patterns required for these calls. For experiments, 10 open source packages and 60 critical commonly used API calls were selected.

The experiment results clearly show the value of our framework. The large number of concrete properties generated are used to check the open source packages. Many errors were found which indicated that the normal programming practice leaves open many possibilities for robustness failures that may cause a commercial software crash in case of exceptions etc. Even in critical packages, many robustness violations have been detected for important API calls. Without our framework, these properties would take many human efforts to write while the correctness is not guaranteed.

5.2 Future Work

Writing concrete properties takes a lot of time and manual effort and makes the use of tools like MOPS difficult and time-consuming. The FSA Generator does the task correctly in matters of seconds.

The FSA Generator can be easily extended to support a lot of other properties in future. Users can specify more rules by writing simple properties with abstract terms. Currently the pattern database present in the framework consists of patterns for C language. More languages can be supported by the framework by adding language specific patterns in the AST-DB. Likewise, more patterns can be added to cover all the possible cases of transitions that may be realized later. The API database can be expanded to include more API
calls. Once all these databases have been modified and expanded, only little modification is required in the automation engine. The current implementation of the engine allows it to be extensible and flexible.

Thus in future, the FSA Generator and correspondingly static checking can be made more powerful resulting in detecting more robustness violations across different language platforms.
List of References


Appendix A

Errors Found in each package

This appendix includes the robustness violations found in the 11 (other than SysV\textit{init} which is already been discussed) packages that were tested for 60 API calls. Figure A.1 and A.2 follows with the information.
<table>
<thead>
<tr>
<th>API</th>
<th># errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>fclose</td>
<td>1</td>
</tr>
<tr>
<td>unlink</td>
<td>1</td>
</tr>
</tbody>
</table>

I) zlib-1.1.3-3

<table>
<thead>
<tr>
<th>API</th>
<th># errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>fgets</td>
<td>1</td>
</tr>
<tr>
<td>malloc</td>
<td>1</td>
</tr>
<tr>
<td>setuid</td>
<td>1</td>
</tr>
<tr>
<td>strchr</td>
<td>1</td>
</tr>
<tr>
<td>write</td>
<td>5</td>
</tr>
</tbody>
</table>

III) rsh-0.17-1

<table>
<thead>
<tr>
<th>API</th>
<th># errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>close</td>
<td>1</td>
</tr>
<tr>
<td>closedir</td>
<td>1</td>
</tr>
<tr>
<td>fflush</td>
<td>1</td>
</tr>
<tr>
<td>malloc</td>
<td>1</td>
</tr>
<tr>
<td>readdir</td>
<td>1</td>
</tr>
</tbody>
</table>

IV) ncompress-4.2.4-33

<table>
<thead>
<tr>
<th>API</th>
<th># errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdir</td>
<td>1</td>
</tr>
<tr>
<td>close</td>
<td>9</td>
</tr>
<tr>
<td>fclose</td>
<td>5</td>
</tr>
<tr>
<td>fflush</td>
<td>4</td>
</tr>
<tr>
<td>fileno</td>
<td>2</td>
</tr>
<tr>
<td>fputc</td>
<td>1</td>
</tr>
<tr>
<td>fputs</td>
<td>4</td>
</tr>
</tbody>
</table>

V) syslogd-1.3.31-3

<table>
<thead>
<tr>
<th>API</th>
<th># errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>fclose</td>
<td>4</td>
</tr>
<tr>
<td>fdopen</td>
<td>2</td>
</tr>
<tr>
<td>fileno</td>
<td>2</td>
</tr>
<tr>
<td>fseek</td>
<td>1</td>
</tr>
<tr>
<td>fstat</td>
<td>1</td>
</tr>
<tr>
<td>putchar</td>
<td>1</td>
</tr>
<tr>
<td>strchr</td>
<td>3</td>
</tr>
</tbody>
</table>

VI) tftp-0.32-4

Figure A.1: Violations found in packages tested
<table>
<thead>
<tr>
<th>API</th>
<th># errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>close</td>
<td>2</td>
</tr>
<tr>
<td>closedir</td>
<td>2</td>
</tr>
<tr>
<td>fclose</td>
<td>16</td>
</tr>
<tr>
<td>fflush</td>
<td>4</td>
</tr>
</tbody>
</table>

VII) sysstat-4.0.7-3

<table>
<thead>
<tr>
<th>API</th>
<th># errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>close</td>
<td>3</td>
</tr>
<tr>
<td>fclose</td>
<td>3</td>
</tr>
<tr>
<td>fflush</td>
<td>3</td>
</tr>
<tr>
<td>fileno</td>
<td>1</td>
</tr>
<tr>
<td>malloc</td>
<td>3</td>
</tr>
<tr>
<td>putchar</td>
<td>5</td>
</tr>
</tbody>
</table>

IX) ftp-0.17-17

<table>
<thead>
<tr>
<th>API</th>
<th># errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>close</td>
<td>2</td>
</tr>
<tr>
<td>fclose</td>
<td>1</td>
</tr>
<tr>
<td>fflush</td>
<td>1</td>
</tr>
<tr>
<td>putchar</td>
<td>1</td>
</tr>
<tr>
<td>setgid</td>
<td>1</td>
</tr>
<tr>
<td>setuid</td>
<td>1</td>
</tr>
</tbody>
</table>

X) traceroute-1.4a12-9

<table>
<thead>
<tr>
<th>API</th>
<th># errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>close</td>
<td>3</td>
</tr>
<tr>
<td>fclose</td>
<td>2</td>
</tr>
<tr>
<td>fflush</td>
<td>8</td>
</tr>
<tr>
<td>fileno</td>
<td>1</td>
</tr>
<tr>
<td>malloc</td>
<td>1</td>
</tr>
</tbody>
</table>

XI) routed-0.17-14

Figure A.2: Violations found in packages tested cont’d
Appendix B

Malloc Use Before Check Error Code Snippets

Appendix B shows examples of codes from several packages that gave *malloc Use Before Check* violations. In total of 14 packages tested, 11 violations were detected.
Figure B.1: Code snippets
routed-0.17-14

```
extern interfaces++;
    ifp = (struct interface *)malloc(sizeof (*ifp));
    memset(ifp, 0, sizeof (*ifp));
    ifp->int_flags = IFF_REMOTE;
    /* can't identify broadcast capability */
    ifp->int_net = inet_netof_subnet(dst.sin_addr);
```

ftp-0.17-17

```
\begin{tabular}{|ll|}
\hline
I & case LOGIN: \\
& \quad if (token()) \{ \\
& \quad \quad if (*aname == 0) \{ \\
& \quad \quad \quad *aname = malloc((unsigned) strlen(tokval) + 1); \\
& \quad \quad \quad (void) strcpy(*aname, tokval); \\
& \quad \quad \} else \{ \\
& \quad \quad \quad if(strcmp(*aname, tokval)) \\
& \quad \quad \quad \quad goto next; \\
& \quad \quad \} \\
& \quad \}

II & if (token() && *apass == 0) \{ \\
& \quad *apass = malloc((unsigned) strlen(tokval) + 1); \\
& \quad (void) strcpy(*apass, tokval); \\
& \quad break;

III & if (token() && *aacct == 0) \{ \\
& \quad *aacct = malloc((unsigned) strlen(tokval) + 1); \\
& \quad (void) strcpy(*aacct, tokval); \\
& \}
\end{tabular}
```

\textbf{Figure B.2}: Code snippets cont’d
Figure B.3: Code snippets cont’d
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>vixie-cron-3.0.1-74</strong></td>
<td></td>
</tr>
</tbody>
</table>
| I | for (j=head; j;j=j->next)  
   if (j->c == c && j->u == u) { return; }  
   /* build a job queue element */  
   j = (job*)malloc(sizeof(job));  
   j->next = (job*)NULL;  
   j->c = c;  
   j->u = u; |
| II | u = (user*)malloc(sizeof(user));  
    u->name = strdup(name);  
    u->cronab = NULL; |
| III | p = (char**)malloc((count+1) * sizeof(char*));  
   /* I for the NULL */  
   for (i=0; i<count; i++)  
     p[i] = strdup(ewp[i]);  
   p[count] = NULL;  
   return (p); |

**Figure B.4:** Code snippets cont’d