Mining Interface Specifications for Generating Checkable Robustness Properties

Mithun Acharya, Tao Xie, Jun Xu
Department of Computer Science
North Carolina State University
Raleigh NC USA 27695
mpachary@ncsu.edu, {xie, junxu}@csc.ncsu.edu

Abstract

A software system interacts with its environment through interfaces. Improper handling of exceptional returns from system interfaces can cause robustness problems. Robustness of software systems are governed by various temporal properties related to interfaces. Static verification has been shown to be effective in checking these temporal properties. But manually specifying these properties is cumbersome and requires the knowledge of interface specifications, which are often either unavailable or undocumented. In this paper, we propose a novel framework to automatically infer system-specific interface specifications from program source code. We use a model checker to generate traces related to the interfaces. From these model checking traces, we infer interface specification details such as return value on success or failure. Based on these inferred specifications, we translate generically specified interface robustness rules to concrete robustness properties verifiable by static checking. Hence the generic rules can be specified at an abstract level that needs no knowledge of the source code, system, or interfaces. We implement our framework for an existing static analyzer that employs push down model checking and apply the analyzer to the well known POSIX-API system interfaces. We found 28 robustness violations in 10 open source packages using our framework.

1 Introduction

As computers penetrate every aspect of our daily life, robustness of software systems is becoming increasingly critical for the dependable operation of computer systems. The IEEE Standard Glossary of Software Engineering Terminology (IEEE STD 610.12-1990) [1] defines *robustness* as the "*degree to which a system or a component can function correctly in the presence of invalid inputs or stressful environment conditions*." Stressful environment conditions may occur in forms such as high computation load, memory exhaustion, process related failures, network failures, file system failures, and slow system response because of the competition among applications for resources. These problems, however rare, should be gracefully handled. Robustness violations often lead to system crashes, leakage of sensitive information, and complete security compromises when the stressful conditions or exceptional inputs are not properly handled.

Stressful environment conditions often occur at interfaces where software systems interact with its environment. It is well known that many robustness failures are due to incorrect exception handling from system interfaces. The exceptional interface values create stressful environment and they should be properly handled. Traditional software testing focuses on correctness of functionality and is often insufficient for assuring the absence of interface-level robustness violations. Robustness testing has been especially conducted to test the robustness of a system [12, 13, 15]. These existing approaches consider the target applications or operating systems as a black box, and send random or exceptional input values through their system input interfaces.

However, robustness testing approaches cannot easily generate implicit return exceptions through system interfaces, which are an important type of sources for robustness problems. To assure the absence of robustness problems related to system interfaces, we can specify robustness properties for system interfaces and statically verify them against a software system. But manually specifying a large number of interface properties for static verification is cumbersome, and requires the knowledge of interface specifications, which are often unavailable or undocumented.

To address these issues, we propose a novel framework to automatically infer the system specific details of interface specifications from program source code. We use a model checker to generate traces related to the interfaces. From these model checking traces, we infer interface specification details such as return value on success or failure. Based on these inferred specifications, which were originally manually specified, our previous work [2] translates generically specified interface robustness rules to concrete robustness properties verifiable by static checking. Hence the generic rules can be specified at an abstract level that needs no knowledge of the source code, system, or interfaces. We implement our framework for an existing static analyzer that employs push down model checking and apply the analyzer to the well known POSIX-API system interfaces.

This paper makes the following main contributions:

- We propose a framework for automatically inferring the interface specifications directly from the program source code and show how the inferred specifications can be used in generating robustness properties for static analysis.

- We implement the framework in an existing static analyzer that employs push down model checking and apply the analyzer to the well known POSIX-API system interfaces. We analyzed 10 Redhat-9.0 packages across 60 APIs. Our framework inferred specifications for 22 of the 60 APIs and could detect 28 robustness violations in the analyzed packages. These numbers are expected to increase as we analyze more packages that use these interfaces.

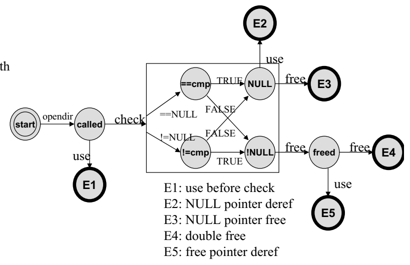
The rest of the paper is organized as follows. In Section 2, we present a simple example to illustrate our framework. Section 3 presents the background for the framework. Section 4 explains the mechanism for generating interface traces using model checking from which specifications can be inferred. Evaluation results are presented in Section 5. Section 6 reviews related work. Section 7 discusses issues in the proposed approach and future work. Section 8 concludes the paper.

2 Example

We present a simple example to illustrate our framework using the `opendir` POSIX-API. The `opendir` POSIX-API opens a directory stream corresponding to the directory name and returns a pointer to the directory stream. On failure, the API returns a `NULL` pointer. The input argument for `opendir` is of type `const char*`. If the program does not have the permission to open the directory, `opendir` returns `NULL` and the global variable `errno` is set to `EACCES` denoting a denied permission. Other error flags are `EMFILE`, `ENFILE`, `ENOENT`, `ENOMEM`, and `ENOTDIR` depending on the error type. These details are documented in the UNIX manual. POSIX-API are widely used and well known. But most interfaces are system or application specific and their specifications are often undocumented. For an application using `opendir` interface to be robust, the `opendir` interface is expected to adhere to certain robustness rules. These rules can be formally specified and statically verified against a software system. Manually specifying such rules for a large

```
...
p = opendir ("\cvs\root\");
...
if ( p == NULL ){ // error path
  ...
  if ( errno == EACCES ){
    ...
    }
  ...
  exit();
  }
else {
  ...
  }
...
```

(a) Example Code snippet for *opendir*



(b) Simplified concrete property *opendir* interface

Figure 1. Example code and a concrete property for `opendir` interface

number of interfaces is cumbersome (there are more than 300 POSIX-APIs and hence thousands of rules), and requires the knowledge of interface specifications. Our framework infers the interface details from the source code and automates the generation of robustness properties.

Figure 1(a) shows an example code snippet using `opendir`. For each `opendir` call in the program, we extract the return value checks (`if ( p == NULL )`), and error flag checks (`if (errno == EACCES)`) associated with the call from the program source code, and output them as traces. For the trace generation, we adapt a compile-time model checker with certain data flow extensions to correctly associate these checks with the call to `opendir`. From the check traces, we derive the majority member and infer specification details such as return values and error flags for the `opendir` interface. For example, if comparison of the return value with `NULL` occurs most of the time, then we discard other comparisons in the traces for the purpose of inference. We use the `gcc` compiler to extract the interface signature and deduce that the input parameter to `opendir` is of type `const char *` and the return variable, `p`, is a pointer type. The signature is used by our framework to infer interface exception values. For example, if the return value `p` is compared to 0, and if the return type is a pointer, then the exception value is `NULL` and not integer 0. In our inference algorithm, we use certain heuristics such as "failure checks are usually followed by a call to `exit` or similar other abort routines" to identify the error paths in conditional checks for a return value.

After the specification details are inferred from the source code, users specify the robustness rules for APIs at a high level using generic keywords such as $check$ (return value check), use (dereference the return pointer), and $free$ (deallocate memory). These keywords are free from interface and source code details. Our framework instantiates these generic rules into concrete properties verifiable by a static analyzer. Figure 1(b) shows a highly simplified concrete property generated for the `opendir` interface. The concrete property governs the correct usage of pointer re-

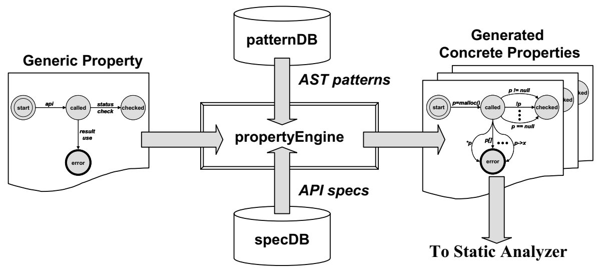


Figure 2. Approach for generation and verification of concrete properties

turn variables. Simplified details (shown in the box) are shown only for one keyword, *check*, which is split into multiple edges and states using return value information from the inferred specification database. The concrete properties for error flags and input parameter checks can be similarly generated for the `opendir` interface. These concrete properties can be used by a static analyzer for statically verifying the robustness of software packages that use the `opendir` interface.

3 Background

In our previous work [2], we proposed an approach that effectively generates interface robustness properties for static verification. The goal was to allow developers to specify robustness rules generically without the knowledge of the system, language, and interface so that these rules can be verified against the system under analysis. To abstract away these details from developers, we make use of two key observations about interfaces and their robustness rules. The first observation is that related interfaces have similar structural *elements* when specified at a certain abstract level. The second observation is that most interface robustness violations are temporal orderings of certain *actions* that could be performed on an interface or its elements.

The high-level overview of our approach is shown in Figure 2. Developers define generic rules at a high level over interface elements and actions, without the details of interfaces and source code. The details of interfaces are stored in a specification database (`specDB`) and the source-code details of interface elements and actions are stored in a pattern database (`patternDB`). The generic rules are translated into concrete properties by a `propertyEngine` that queries `specDB` for interface-level information and `patternDB` for source-code level, programming-language specific information. Using this approach, we found around 200 robustness problems in 10 Redhat-9.0 packages. Specifying the interface specification requires much human effort and interface knowledge. In this paper, we present our framework to automatically infer the specification database, which was originally specified manually. We next formally define an interface specification in Section 3.1. Section 3.2 describes the specification database (especially for the POSIX-APIs), which was manually specified in our previous approach [2] and will be inferred by our new framework. We also briefly describe the pattern database used by our approach. In Section 3.3, we provide a simple example and illustration of a generic rule and its corresponding concrete property.

3.1 Interface Characterization

A set of interfaces (such as functions to be invoked) implemented for a specific purpose has similar structural details at a high level. We characterize an interface with its structural elements (such as function parameters and returns) and actions that can be performed on them (such as checking a function's return for failure). The characterization allows us to systematically store the interface and language patterns for these interfaces in a database. For a given interface, the `propertyEngine` can query the database on the keywords of elements or actions to get low-level details.

For any interface $i \in \mathcal{I}$, where $\mathcal{I}$ is a related family of interfaces, we define an interface specification as $spec(i) = \{is(i), rs(i), ss(i), \mathcal{R}, \mathcal{S}, \mathcal{Z}\}$, where $is(i)$ is the set of input parameters passed to the invocation of i, $rs(i)$ is the result set, the set of variables that store the return values of interface execution and $ss(i)$ is the status set, the set of variables that store the failure status or type of failures of the interface. Any variable $v \in is(i) \bigcup rs(i) \bigcup ss(i)$ is called the *element* of i. $\mathcal{R}$ is a mapping from $rs(i)$ to $\mathcal{Z}$, while $\mathcal{S}$ is a mapping from $ss(i)$ to $\mathcal{Z}$, where $\mathcal{Z}$ holds the values that members of $rs(i)$ and $ss(i)$ would assume on success or failure of interface execution. For a related family of interfaces, $\mathcal{I}$, we define an $action$ set as a set of actions that can be performed on the interface itself or its elements.

For example, $\mathcal{I}$ could be POSIX-API interfaces and $i \in \mathcal{I}$ could be `malloc`. Before a statement such as `p = malloc(x)` is executed in a program, $is(\texttt{malloc})$ is $\{x\}$, $rs(\texttt{malloc})$ is $\emptyset$ and $ss(\texttt{malloc})$ is $\emptyset$. After the statement execution, $rs(\texttt{malloc})$ is $\{\texttt{p}\}$ and $ss(\texttt{malloc})$ is $\{\texttt{p}\}$. For `malloc`, the return and the failure/success indicator are the same. If the `malloc` call succeeds, `p` is a memory pointer (say, `mptr`) and then $(\texttt{p}, \texttt{mptr}) \in \mathcal{R}$. If it fails, `p` assumes value `NULL` and $(\texttt{p}, \texttt{NULL}) \in \mathcal{R}$. Because the result set and status set are the same for `malloc`, we have $\mathcal{S} = \mathcal{R}$. The set $\mathcal{Z} = \{\texttt{mptr}, \texttt{NULL}\}$ holds the success/failure indicators for the `malloc` API. Some example actions that could be performed on `malloc` interface elements are `check` (which checks the return against `NULL`) and `use` (which dereferences the return pointer). The interface specification does not have any system or language specific details. We next describe the `specDB` and the `patternDB` and show how concrete interface robustness properties can be generated from generic rules defined over elements and actions.

Table 1. Selected entries from the `specDB` for POSIX-APIs (simplified for presentation)

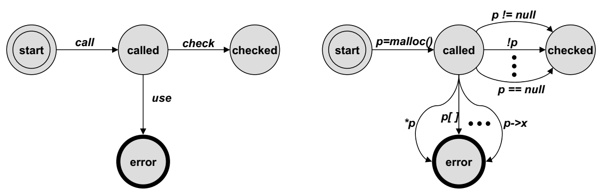
API	parameter list	return type	return value		errno
			on success	on failure	
chmod	const char * path , …	int	0	-1	EPERM, …
open	const char * pathname, …	int	fd	-1	EEXIST, …
malloc	size_t size	void *	pointer	null pointer	
fsetpos	FILE * stream , …	int	0	-1	EBADF, …
remove	const char * pathname	int	0	-1	EFAULT, …

3.2 Specification and Pattern Database

The Specification Database (`specDB`) defines the type of all $v \in spec(i)$, the mappings $\mathcal{R}$ and $\mathcal{S}$, the set $\mathcal{Z}$ for all $i \in \mathcal{I}$, and the $action$ set for $\mathcal{I}$. The rows of the `specDB` are interface names and the columns represent the specification details of an interface. The set of column names forms the abstraction that characterizes all the interfaces in that family. As an example, the entry for `setuid` in the POSIX-API `specDB` lists the type of its input parameters as `int`, the return values to be 1 on success and -1 on failure, and the error flag to be set as `EPERM` on failure. Table 1 shows several typical interface specifications for POSIX-APIs in a simplified form.

In our previous work [2], we inspected 280 POSIX-APIs to arrive at a common abstraction to characterize all the interfaces. We defined the types for the variables in the sets of $is(i)$, $rs(i)$ and $ss(i)$. We established the mappings $\mathcal{R}$ and $\mathcal{S}$ and the set $\mathcal{Z}$ for each of the 280 interfaces. For an interface i that sets error flags on failure, the global variable $errno \in ss(i)$. The corresponding error status resides in $\mathcal{Z}$. For example, if the API `setuid` fails, then $errno \in ss(setuid)$ is set to `EPERM` $\in \mathcal{Z}$.

The Pattern Database (`patternDB`) contains language-specific details of the elements for all $i \in \mathcal{I}$ and actions in the $action$ set of $\mathcal{I}$. We use the Abstract Syntax Tree (AST) notation for storing patterns. The patterns for the use action depend on the data type of the return and the patterns for the $check$ action depend both on the data type of the return and the success/failure indicator/type. The `patternDB` stores patterns for all members of the $action$ set for each data type, success/failure indicators, etc. As an example, the $check$ action against the `zero` patterns for an integer variable `p` would be `(p==0)`, `(p!=0)`, `(p)`, `(!p)`, and so on. The *call* patterns for the `malloc` API would be `i=malloc(...)`, `if(i=malloc(...))!=NULL`, and so on. The content in the `patternDB` is manually specified but the manual effort is one time; the content of the `patternDB` can be reused as long as the software packages to be checked are written in the same programming langauge. In the current implementation of the framework, we specify the content in the `patternDB` for the C language but patterns for other languages can also be similarly specified.



(a) *check* should always precede *use* generic property

(b) Concrete *UseBeforeCheck* property for the *malloc* API call

Figure 3. Generic *UseBeforeCheck* **property and the corresponding concrete property for** `malloc` **API**

3.3 Robustness Property Generation and Verification

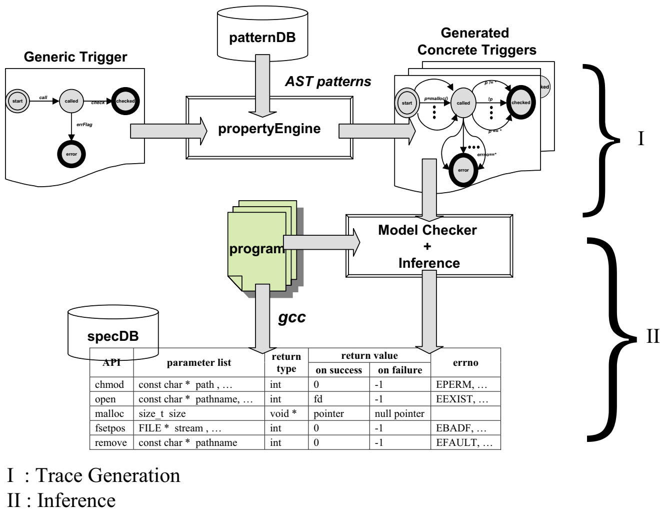
Generic rules for an interface $i \in \mathcal{I}$ are defined over the members of the $action$ set of $\mathcal{I}$. A generic rule is some ordering constraints on the members of the $action$ set.

We use a Finite State Machine (FSM) to graphically represent a generic rule. The FSM has a start state and an error state as well as other user-defined states. A sequence of actions that violates the robustness property represented by the FSM takes the FSM to the error state. The edges of the FSM are members from the $action$ set. For example, for the `malloc` interface, the use action should always be preceded by the $check$ action. The FSM for such a rule is shown in Figure 3(a). Generic robustness rules are currently manually specified.

To generate the concrete property for `malloc`, `propertyEngine` queries the `specDB` to obtain details about `malloc` and learns that the return type of `malloc` is a pointer on success and `NULL` on failure. Based on this information, the `propertyEngine` constructs a query to the `patternDB` that comprises the keyword $check$, the data type of the return variable, and values on success and failure (being a pointer in this case). The `patternDB` processes this query and returns patterns for all the possible ways that a pointer variable can be checked against `NULL` (or not `NULL`) (`if(p==NULL)`, `if(p)`, `if(!p)`, etc.). The `propertyEngine` expands the generic keyword $check$ to language and interface specific patterns. The same procedure applies to the keyword *call* (`if(p=malloc(...))!=NULL`, `p=malloc(...)`, etc.) and use (`p->x`, `*p`, `p[x]`, etc.). The generated concrete property is shown in Figure 3(b). These concrete properties can be used by static analyzers [4–6, 14] to detect robustness violations in software packages.

4 Framework

This section presents our framework for inferring interface specifications from program source code. For our preliminary experiments and results [2], we manually gener-



API	parameter list	return type	return value		errno
			on success	on failure	
chmod	const char * path , …	int	0	-1	EPERM, …
open	const char * pathname, …	int	fd	-1	EEXIST, …
malloc	size_t size	void *	pointer	null pointer	
fsetpos	FILE * stream , …	int	0	-1	EBADF, …
remove	const char * pathname	int	0	-1	EFAULT, …

Figure 4. Framework for Inferring Interface Specifications

ated the specification database for more than 280 POSIX-APIs. Table 1 shows several typical interface specifications for POSIX-APIs in a simplified form. Informally stated, the `specDB` characterizes the input, output, and error specifications for all the POSIX-APIs. The entries in the database show the input parameters for API calls, return value types, status on success/failure, and the different error flags set on failures. We built the `specDB` database for 280 POSIX-APIs manually by inspecting the UNIX manual pages. The database can be used by our framework as well as other research projects.

Databases similar to that for POSIX-APIs can be built for different families of interfaces. Although it is a one time effort, it is a tedious process. POSIX-APIs are widely used and well known. Their specifications can be found in the UNIX manual pages. But most interfaces are system or application specific and their specifications are often undocumented. Interfaces have different return values on success and failure, error flag values, proper checking routines, and correct usage rules. Table 2, for instance, classifies the 280 POSIX-APIs into 11 classes based on their return values. For example, `closedir` belongs to the class $0 : -1$, which means it returns 0 on success and -1 on failure. `malloc` belongs to the class $p : np$, which means it returns a pointer (p) on success and a `NULL` pointer (np) on failure. By manually inspecting source code, we cannot easily infer these interface details. In this section, we describe our framework to derive these specifications automatically from the program source code by using inference techniques on model checking traces.

The high-level overview of our framework is shown in Figure 4. There are two main components in our framework as shown in the figure. Section 4.1 describes the generation of traces using push-down model checking. Section 4.2 de-

Table 2. API classification based on return values

No.	API class	Sub-classes	Examples
I	0:-1	0:-1	closedir
		x, 0:-1	mblen
		fd, flag, fd_owner, signal, 0:-1	fcntl
II	0, 1, -1:-1	-1, 1, 0:-1	sysconf
		1, 0:-1	readdir
III	0:EOF	0:EOF	fflush
		x, 0:EOF	scanf
		0:EOF, undefined	fclose
IV	0:x	0:x	ttyname_r
		x, 0:0	strftime
		x, 0::	isupper
		0:0	sem_wait
		0:any_number	setvbuf
		0, non_zero : 0, non_zero	setjmp
		second, 0::	alarm
		x, p, 0:-x, 0	strtod
V	1,0::	1, 0::	isatty
VI	np, -1, 0, x:np, -1, 0, x	np, -1, 0, x:np, -1, 0, x	wctomb
VII	p:np	p:np	malloc
		p, np:p, np	strtok
		p, np:np	realloc
VIII	x:-1	x:-1	ftell
		::-1	feof
		no_return:-1	execve
		x, -1:-1	pathconf
IX	x:EOF	x:EOF	putchar
X	x, EOF:np	x, EOF:np	getchar
XI	x:negative	x:negative	printf

scribes how the specification database can be inferred from the generated traces.

4.1 Trace Generation

In this section, we describe how model checking can be used for generating traces. The key idea is to force the model checker to output interface $action$ traces along each execution sequence in the program. *Trigger* automata (described in Section 4.1.2 and 4.1.3) are used for this purpose and can be generically specified by the users. The program statements in these traces contain interface $actions$ that are necessary for `specDB` inference.

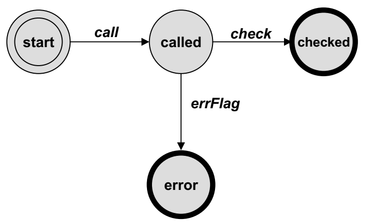
4.1.1 Generation of Traces with Model Checking

Push-down model checking [11] has been used for verifying programs against a certain class of temporal properties [6]. The temporal properties are specified as a Finite State Machine (FSM) with start and error states. The edges of the FSM are program statements. The FSM captures all sequences of program statements that take it from the start state to one of the error states. In other words, sequences of program statements that cause property violations take the FSM from the start state to one of the error states. This FSM is composed with the Push Down Automata (PDA) constructed for the C program from its Control Flow Graph (CFG) to form a new PDA. The new PDA is model checked to see if there are any paths in the program that take the new PDA to an error configuration.

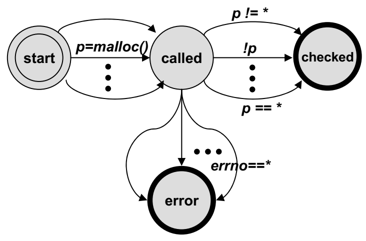
The model checking procedure systematically explores different execution sequences in the program to verify the property specified by the FSM. Since the PDA transition happens on a program statement, the push-down model checking procedure encounters different program statements during this exploration. The property FSM determines what execution sequences and corresponding program statements need to be reported by the model checking procedure. Hence by manipulating the property FSM, and instrumenting the model checker, we force the model checking procedure to selectively and conservatively output different program traces related to interface $actions$. In the next section, we show how trigger automata can be used to force the push-down model checking to output interface $action$ traces.

4.1.2 Specification of Generic Triggers

Interface $actions$ are often temporally ordered. For example, after interface $call$, $check$ should precede use. Violations of these temporal orderings lead to robustness problems. When generic rules are converted to concrete properties, the patterns for these interface actions can be queried from the `patternDB`. The FSM representing the concrete property has details about the return exceptions, error flags,



(a) Generic Trigger Automata– Causes the model checker to output *check* and *errFlag* traces



(b) Concrete Trigger Automata for *malloc* – Causes the model Checker to output *check* and *errFlag* traces for *malloc* API

Figure 5. Trigger Automata - Forces the model checker to ouput interface $action$ traces along each execution path

and so on. The `propertyEngine` queries the `specDB` to gather these details. The goal of our framework is to infer the `specDB` directly from the program source code.

Formally, trigger automata are Finite State Machines in which the transition to the final state happens on an interface $action$. Unlike the FSMs representing generic properties, these automata do not define any robustness rules. Instead, the trigger FSM cause the push-down model checking procedure to output interface $action$ traces. For example, Figure 5(a) defines a generic trigger automata used for forcing the model checker to output the $check$ and $errFlag$ (`errno` compared with error flags) traces for the given API.

4.1.3 Generation of Concrete Triggers

Figure 5(b) shows the corresponding concrete trigger for the `malloc` interface automatically generated for trigger automata shown in Figure 5(a), which can be specified by users generically. The `propertyEngine` queries the `patternDB` to convert a given generic trigger to a concrete trigger. However, unlike the translation of generic rule to concrete property, this translation does not use the `specDB` to get the interface details such as exceptional values and error flags. Instead, these details are replaced by wild cards. For example, in Figure 5(b), any comparison of the return value of the `malloc` API takes the automata to its final state. These concrete triggers force the model checking procedure to output interface $actions$ along each execution sequence in the program. For example, the concrete trigger in Figure 5(b) forces the model checker to output $check$ and $errFlag$ traces for the `malloc` API. We illustrate how $check$ traces are generated using the concrete trigger shown in Figure 5(b). Any execution sequence having a call to the `malloc` interface followed by a check on `malloc` return value causes the concrete trigger automata to transit to the final state. The model checker outputs program statements for each such execution sequence. $check$ traces can be generated from these execution sequences as described in the next section. The generation of $errFlag$ traces is similar.

4.2 Inference of Specification Database from Traces

The push-down model checking process generates sequences of program statements that take the concrete trigger automata from the start state to the final state. For inferring interface specifications, we are interested in the last statement that caused the transition to the final state in each sequence of program statements generated by the model checking procedure. The program statement that causes the FSM to transit to the final state is some *action* performed on the interface. The set of last program statements of each execution sequence constitutes the traces generated by the model checker. An inference algorithm is used on these traces to infer the specification details of a given interface. For example, while generating the $check$ trace for the `malloc` API, the execution sequences that take the trigger automata from the initial state to the final state are those in which the `malloc` interface is invoked and its return value is checked. Each program statement that checks the return value of the `malloc` API (for example, `if(p=malloc()==NULL)`) constitutes the $check$ trace for `malloc`. Our inference algorithm runs on these traces and has the following steps:

- For each conditional construct in the trace that checks the return value, we determine the success path and the failure path. For example, after `p = malloc(...)`, the TRUE branch of the conditional check, `if (p == NULL)`, is the failure path. Failure checks are usually followed by a call to `exit` or similar other abort routines.

- When we cannot determine the success path and the failure path using the preceding abort routine heuristic, we use other heuristics to distinguish return values on success and failures. One heuristic is that failure-value checks occur more often than success-value checks. For example, `setuid` returns 0 on success and -1 on failure. Then likely checks in a program would be either `if(setuid(0)==-1)` or `if(setuid(0)!=0)` instead of `if(setuid(0)==0)`. Another heuristic is that failure values are mostly negative numbers (or `NULL`) while success values are mostly non-negative numbers.

- We use the `gcc` compiler to obtain the interface signature for a given interface. The interface signature comprises the input parameter types and the return type. We use the return type to infer the values on interface success and failure. For example, if the return value `p` is compared to 0 on the failure path, and if the return type is a pointer, then exception value is `NULL` and not integer 0. In addition, we can deduce that the interface should return a pointer on success.

- Each conditional construct in the trace is now replaced by a triplet (x, y, z), where $x \in \{!=, =, <, >, <=, >=\}$, y is a constant and z is either $PASS$ or $FAIL$. For example, the conditional construct `if((p=malloc(...)) == NULL){...exit();}` is replaced by $(=, \texttt{NULL}, FAIL)$

- We pick the majority members in the triplets to determine the return values on success and failure. We use a similar algorithm to infer error flags from the $errFlag$ traces. In particular, we observe what values are compared with `errno` most of the times after the interface is invoked. For the $errFlag$ traces, we assume that no system or function call exists between the interface call and `errno` check with error flags.

Specification details are inferred from the traces generated for each interface used in the source code. The trace size is reduced by missing checks for return values (robustness violation) and sparse usage of interfaces in a package. Aliasing and improper checking due to programmer errors lead to noise in the trace. By analyzing more packages that use these interfaces, we can address the problem of noise and reduced trace size.

5 Evaluation

We have applied our framework for inferring the `specDB` for the POSIX APIs. We used our framework to analyze open source packages written in `C` mostly from the Redhat-9.0 distribution. In our experiments, we used a Pentium IV machine with 2.8GHz processor speed and 1GB RAM running on the Fedora Core 3 2.6.9-1.667smp kernel. In the experiments, we selected 10 widely used open source packages from the Redhat-9.0 distribution; these 10 packages include near 100K lines of `C` code. For push-down model checking, we used a publicly available static analyzer called MOPS [5, 6], which detects control-flow errors at compile time. It constructs a Push Down Automaton (PDA) for a `C` program from its Control Flow Graph (CFG). It then generates a new PDA by composing the property FSM to be checked and the program PDA. The new PDA is model checked [11] to see if there is any path in the program that takes the new PDA to an error configuration. If there exists such a path in the program, the static checker reports

Table 3. Inference results for 7 APIs with largest trace size across 10 open source packages

	fopen	fdopen	getenv	getpwnam	malloc	open	opendir
ftp-0.17-17	✓	✓	✓	✓	✓	✗	✗
ncompress-4.2.4-33	✗	✗	✗	✗	✗	✓	✓
routed-0.17-14	✓	✓	✗	✗	✓	✓	✗
rsh-0.17-14	✓	✗	✓	✓	✓	✓	✗
sysklogd-1.3.31-3	✓	✓	✗	✗	✓	✓	✗
sysstat-4.0.7-3	✓	✗	✗	✗	✓	✓	✓
SysVinit-2.84-13	✓	✓	✓	✗	✓	✓	✓
tftp-0.32-4	✗	✓	✗	✓	✗	✓	✗
traceroute-1.4a12-9	✓	✗	✗	✗	✓	✗	✗
zlib-1.1.3-3	✓	✗	✗	✗	✗	✗	✗
Inferred	✓	✓	✓	✓	✓	✓	✓
Trace Size	100	13	15	6	70	116	8

✓ : Specification inferred
✗ : Specification not inferred

the path as the error trace that violates the concrete robustness property. We write extensions to the push down model checking procedure enabling it to output interface action traces determined by the triggers.

The generic triggers can also be data-flow sensitive, i.e., they are dependent on the value of the return variable along different execution paths. Because the basic MOPS static checker is data-flow insensitive, it assumes that a given variable might take any value. Therefore, it assumes that both branches of a conditional statement may be taken and that a loop may execute anywhere between zero to infinite iterations. Because exception handling procedures are usually characterized by conditional constructs that check the return value of an API call, we write extensions to the static analysis procedure to make it track the value of variables that take the return status of an API call along different branches of conditional constructs. For each possible execution sequence, our extensions associate a value to the variable that is being tracked using pattern matching. The concrete triggers (in the form of FSMs) generated by our framework are given to the static analyzer enhanced with our trace generation capabilities and data flow extensions.

Effectiveness: A user only needs to specify a small set of generic triggers at a high level. The `propertyEngine` automatically generates hundreds of concrete triggers for 280 POSIX APIs. For `specDB` inference, we selected 60 critical API calls that are mainly used for memory management, file and string I/O, permission management, setting privileges, and spawning processes. We then generated concrete triggers for them using our `specDB` inference framework. For these 60 APIs, more than 100 concrete triggers were automatically generated by the `propertyEngine` and they were used against 10 Redhat-9.0 open source packages for specification inference.

Usefulness: Table 3 presents the inference results for 7 APIs across 10 packages. We selected 7 APIs that gave largest total trace size with the 10 selected packages. The "Inferred" row in the table specifies if the property could be inferred from the 10 packages. An API specification is said to be inferred from the analysis of a set of packages, if it can be inferred from at least one package in the set. The last row in table shows the trace size for the 7 APIs across 10 analyzed packages. Of the 60 APIs, we could successfully infer the specifications for 22 APIs. Hence just by analyzing 10 packages, we could infer more than a third of the critical APIs used in these packages. We expect the number of APIs with inferred specifications to increase as we analyze more packages. *check* traces are required to infer the return values on success and failure for a given interface. Missing interface return checks (robustness violation) decrease the precision of the inference because of the reduction in the trace size. Sparse usage of a given interface in a package also leads to reduced inference precision.

Table 4. Analysis results for the `rsh-0.17-14` package

API	**#errors**	**Inferred**
fgets	1	✓
malloc	1	✓
setuid	1	✗
strchr	1	✓
write	5	✓

False negatives and positives: Table 4 presents the total number of robustness violations that our tool found for the `rsh-0.17-14` package employing the approach shown in Figure 2, along with inference results. Of the 60 analyzed APIs, 5 APIs gave violations with this package leading to 9 robustness bugs. Detailed results were reported elsewhere [2]. After analyzing the 10 packages, specifications could be inferred only for 4 of the 5 APIs that caused robustness problems. Hence 8 out of 9 robustness bugs in `rsh-0.17-14` can be detected with 4 API specifications inferred from all the 10 packages. The false negative ratio can be decreased by analyzing more packages and inferring more specifications.

False positives can occur because of wrong inference. For instance, if a programmer compares the return value of an interface against a wrong value in the program most of the times, incorrect return values can be inferred for that interface. The resulting incorrect concrete properties lead to false positives during static verification. In the 10 packages we analyzed, we did not find any instances of false positives.

Efficiency: We restricted the model-checking-trace generation to be intra-procedural. We believe that such a design choice can still lead to high inference precision because most of the interface $actions$ do not cut across procedural boundaries. Of the 60 analyzed APIs across 10 open source packages, a maximum 19 APIs gave violations with the `SysVinit-2.84-13` package when analyzed with the approach [2] shown in Figure 2. It took less than 150 seconds to generate the traces across 19 APIs for the `SysVinit-2.84-13` package and infer their specifications from the traces.

6 Related Work

Engler et al. [9] infer bugs by statically identifying inconsistencies from commonly observed behavior. Being motivated by their approach, we statically infer interface specifications from program source code through inference. In addition, our inferred specifications are combined with generic robustness rules to produce concrete robustness properties for static verification.

Various approaches have been developed to dynamically infer properties for a program and statically or dynamically check the program against the inferred properties. For example, Ernst et al. [10] developed the Daikon tool to infer operational abstractions from test executions. Nimmer and Ernst [16] then feed these inferred operational abstractions to a static verification tool in order to filter out inferred operational abstractions that are not universally true. Xie and Notkin [18] feed the inferred operational abstractions to a test generation tool for finding their violations dynamically; generated tests that cause the violations are selected for inspection. Yang et al. [19] infers temporal properties from program executions and then feed the inferred properties to a static verification tool in order to detect their violations for finding bugs. All the preceding approaches use dynamic inference techniques and then check inferred properties with static or dynamic verification whereas our approach uses static inference techniques and verify inferred properties with static verification.

Static compiler analysis has been widely used to find bugs and security holes in source code. The Meta-Compilation (MC) Project uses programmer written compiler extensions to statically find bugs in operating systems [3, 8] and cache protocols. MOPS [6] is a control flow sensitive static checker that checks for certain vulnerable system-call sequences in the program. MOPS, however is data flow insensitive. Tools like SLAM [4] and BLAST [14] are static analysis tools based on theorem proving and model checking boolean abstractions of the program with iterative refinements. These tools do not hide the interface and source code level details from the user. Our proposed framework and its implementation could be easily adapted to effectively generate interface robustness properties for these tools.

Dwyer et al. [7] proposed a pattern-based approach to the representation of property specifications for finite state verification that can accommodate properties typically specified with temporal logic or regular expressions. Developers can use this representation to write generic interface robustness rules. We address the problem of effectively translating these generic rules into concrete properties across many system specific interfaces, hiding the interface and source code level details from the user.

7 Discussion and Future Work

We restricted the model-checking trace generation to be intra-procedural. We believe that such a design choice can still lead to high inference precision as most of the interface $actions$ do not cut across procedural boundaries. Our analysis does not handle aliasing. Aliasing, missing return checks, buggy usage of interfaces can lead to noise in the generated traces. We can address these shortcomings by analyzing more packages thereby increasing the trace size and inference precision. When we analyzed the 10 open source packages using the manually specified `specDB` with the approach [2] shown in Figure 2, we found 188 robustness violations. Using the specification database inference framework proposed in this paper on these 10 packages, we could detect 28 out of 188 robustness violations automatically. The number of bugs detected would increase as the number of packages analyzed increases. We selected the POSIX-APIs for our experiments as their specifications were already documented and available. Using these specifications, we could measure the inference precision of our framework. We plan to analyze more packages and interfaces in the future using our framework.

We inferred specifications from static traces generated by push-down model checking. Dynamic traces generated during execution time can also be used by our inference algorithm. However, test cases are required for generating dynamic traces and they might not exercise all the execution sequences in the source code.

We can extend the framework proposed in this paper to infer properties from the source code. Many application-specific correctness rules govern robust and secure operations of software systems; but these rules are often not documented by the developers. While robustness properties are defined over interfaces, security properties involve multiple system calls. An important class of security properties dictate how a system call or a set of system calls can be used in the program. For example, if the `execl` system function is called to execute a user program with an immediately preceding `setuid(0)`, the user program might get a root privilege to the system. Like robustness properties, most security properties can be defined by certain temporal orderings of system calls.

Intra-procedural analysis is sufficient to extract most robustness properties from source code. But many security properties that dictate the ordering of system calls cut across procedural boundaries. For a given set of system calls, we could use the inter-procedural push down model checking procedure of our static analyzer to generate program traces. Security properties could be mined from these traces. We plan to extend our framework to infer these new types of properties. Program slicing techniques [17] can be used to reduce the trace size. Program slicing causes the model checker to output only the program statements that are relevant to the set of interfaces or system calls under consideration. The application of program slicing reduces the trace size and increases the precision of property inference. Multiple packages can be analyzed to increase the trace size if system calls under consideration are sparsely used in the package being analyzed.

8 Conclusions

We have proposed a novel framework to automatically infer system specific interface specifications from program source code. We used a model checker to generate traces related to the interfaces. We inferred interface specification details such as return value on success or failure from these model checking traces. These specifications were used for translating generically specified interface robustness rules to concrete robustness properties verifiable by static checking. Hence the generic rules can be specified at an abstract level that needs no knowledge of the source code, system or interfaces. We implemented our framework for an existing static analyzer that employs push down model checking and applied it to the well known POSIX-API system interfaces. We found 28 robustness violations in 10 open source packages using our framework.

References

[1] *IEEE Computer Society, IEEE Standard Glossary of Software Engineering Terminology, IEEE STD 610.12-1990.* December 1990.

[2] M. Acharya, T. Sharma, J. Xu, and T. Xie. Effective generation of interface robustness properties for static analysis. In *Proc. IEEE/ACM International Conference on Automated Software Engineering*, 2006.

[3] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *Proc. IEEE Symposium on Security and Privacy*, pages 143–159, 2002.

[4] T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proc. Workshop on Model Checking Software*, pages 103–122, 2001.

[5] H. Chen, D. Dean, and D. Wagner. Model checking one million lines of C code. In *Proc. Network and Distributed System Security Symposium*, pages 171– 185, 2004.

[6] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *Proc. ACM Conference on Computer and Communications Security*, pages 235–244, 2002.

[7] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *Proc. International Conference on Software Engineering*, pages 411–420, 1999.

[8] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. USENIX Symposium on Operating Systems Design*, pages 1–16, 2000.

[9] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proc. ACM Symposium on Operating Systems Principles*, pages 57–72, 2001.

[10] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Eng.*, 27(2):99–123, 2001.

[11] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking push down systems. In *Proc. International Conference on Computer Aided Verification*, pages 232–247, 2000.

[12] J. Forrester and B. P. Miller. An empirical study of the robustness of Windows NT applications using random testing. In *Proc. USENIX Windows Systems Symposium*, pages 69–78, 2000.

[13] J. Haddox, G. Kapfhammer, C. Michael, and M. Schatz. Testing commercial-off-the-shelf software components. In *Proc. International Conference and Exposition on Testing Computer Software*, 2001.

[14] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *Proc. Workshop on Model Checking Software*, pages 235–239, 2003.

[15] P. Koopman and J. DeVale. The exception handling effectiveness of POSIX operating systems. *IEEE Trans. Softw. Eng.*, 26(9):837–848, 2000.

[16] J. W. Nimmer and M. D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *Proc. Workshop on Runtime Verification*, 2001.

[17] M. Weiser. Program slicing. In *Proc. International Conference on Software Engineering*, pages 439–449, 1981.

[18] T. Xie and D. Notkin. Tool-assisted unit test selection based on operational violations. In *Proc. IEEE International Conference on Automated Software Engineering*, pages 40–48, 2003.

[19] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining temporal API rules from imperfect traces. In *Proc. International Conference on Software Engineering*, pages 282–291, 2006.