Automatic Inference of Interface Properties from Program Source Code

Mithun Acharya
Advisors: 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

Our research proposes a novel framework to automatically infer system-specific interface properties from program source code using static model-checking traces.

Area: Software Engineering, *sub-area*: Software Verification

1. INTRODUCTION

Robustness, security, and performance of software systems are governed by various temporal properties related to system interfaces. Violations of these properties often lead to system crashes, security compromises, and performance degradation. Static verification has been shown to be effective in checking temporal properties related to system interfaces. But manually specifying these properties is cumbersome and requires the knowledge of interface specifications. We design and implement a novel approach [1] to effectively generate a large number of concrete interface robustness properties for static verification from a few generic, high-level user specified robustness rules for exception handling. Interface specifications, required for the generation of concrete properties, and many system-specific properties are often not documented by developers. Our research proposes a novel framework to automatically infer system-specific interface specifications [2] and temporal properties from program source code. We use a model checker to generate static traces related to the interfaces. The interface specifications and temporal properties are inferred from these traces using statistical analysis and certain heuristics. We report our initial experience of applying our framework on the POSIX-APIs used in Redhat-9.0 open source packages. We implement our ideas in an existing static analyzer that employs push-down model-checking.

The rest of the paper is organized as follows. Section 2 presents our approach for generating interface robustness properties. Section 3 presents our proposed framework for generating interface traces using model checking, from which interface specifications and properties can be inferred. Section 4 describes preliminary results. Section 5 presents related work and Section 6 concludes.

FSE '06 Oregon, Portland USA

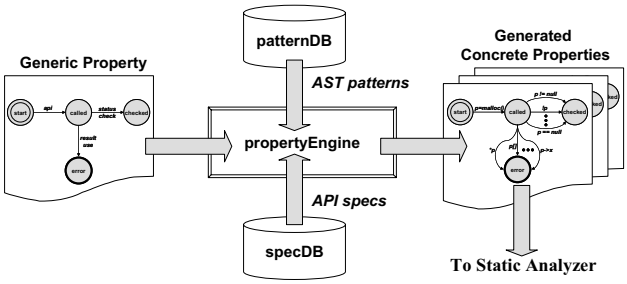
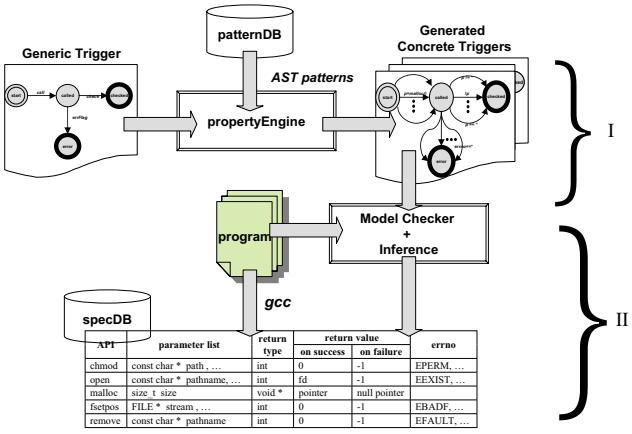


Figure 1: A Framework for Generating Concrete Properties

2. PROPERTY GENERATION

Our goal is to allow developers to specify robustness rules generically without the knowledge of the system, language, or interfaces. These rules can then be verified against the system under analysis. To abstract away these details from developers, we use two key observations about interfaces and their robustness rules. The first observation is that related interfaces have similar structural *elements* (such as function parameters, returns, return values on success/failure, and error flags) when specified at a certain abstract level. The second observation is that most interface robustness violations are temporal orderings of certain interface *actions* (such as invoking an interface, checking interface return for success/failure, dereferencing interface return value, aliasing interface return value, and passing interface return value to a function) that could be performed on an interface or its elements.

The overview of our approach [1] is shown in Figure 1. Developers define generic rules at a high level over interface elements and actions, without the details of interfaces and source code. Generic rules are Finite State Machines (FSM) whose transition edges are interface *actions*. 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`) using the Abstract Syntax Tree (AST) notation. The generic rules are translated into concrete properties by a `propertyEngine` that queries `specDB` for interface-level information and `patternDB` for source-code details of interface elements and actions. The `pattern-DB` is a constant file specific to each programming language and contains the source code information for different language operations (such as dereference and check) that can be performed on simple and derived data types. The `pattern-DB` can also be built for languages such as C++ and Java. The details of our approach are given elsewhere [1].



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, …

I : Trace Generation
II : Inference

Figure 2: Framework for Inferring Interface Specifications

3. PROPOSAL AND CONTRIBUTIONS

For our preliminary experiments and results [1], we manually generated the specification database for more than 280 POSIX-APIs. 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. The return values of such interfaces on success and failure, error flag values, proper checking routines, or correct usage of a given interface or a set of interfaces cannot be immediately inferred from inspecting the source code.

We next concisely describe our approach to derive these specifications automatically from the program source code by using inference techniques on model checking traces. The key idea is to force the model checker to output interface *action* traces along each execution sequence in the program. *Trigger* automata 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. Trigger automata are Finite State Machines in which state transitions happen on selected interface $actions$. Unlike the FSMs representing generic rules, these automata do not define any interface rules. Instead, the trigger FSM causes the push-down model checking procedure to selectively and conservatively output interface $action$ traces. Interface specifications are inferred from these $action$ traces. Our inference algorithm uses certain heuristics such as "failure checks are usually followed by a call to `exit` or similar other abort routines", "interface returns are usually checked against failure values", and "failure return values are usually negative numbers". The high-level overview of our framework is shown in Figure 2.

We can extend our proposed framework to infer application-specific interface properties (from the source code), which are often not documented by the developers. These properties, if violated, can have robustness, security, performance, or race-condition implications. Interface robustness properties, for example, are often intra-procedural in nature. Such properties can be inferred from analyzing intra-procedural model-checking traces. For a given interface, we can write trigger automata to cause the model checker output traces involving interface $actions$ and other program statements that get invoked before, after, or between two interface actions. From these traces, we can infer how other interfaces, function calls, parameter variables, success/failure checks, and other related program statements interact with the given interface. By implementing simple data flow extensions to the push-down model-checking procedure, we could infer basic data-flow-sensitive interface properties.

An important class of security properties dictate how an interface or a set of interfaces can be used in the program [7,18]. Like robustness properties, most security properties can be defined by certain temporal orderings of interface $actions$. Intra-procedural analysis is sufficient to extract most robustness properties from source code. But many security properties that dictate the ordering of interfaces cut across procedural boundaries. For a given set of interfaces, we use the inter-procedural push-down model-checking procedure of our static analyzer to generate program traces. Inter-procedural properties can be mined from these traces. The users specify a set of interesting APIs to be analyzed. Triggers are generated for these APIs from which properties are statistically inferred. The inferred properties can be fed back to the model checker for verification against other packages. Stronger inference can be done based on the number of times the properties are violated or satisfied during this verification process and also on the knowledge of locations where the property was violated. Program slicing techniques [20] can be used to reduce the trace size. Program slicing causes the model checker to output only those program statements that are relevant to the set of interfaces under consideration. Slicing reduces the trace size and increases the precision of property inference. Multiple packages can be analyzed to increase the trace size if interfaces under consideration are sparsely used in the package being analyzed.

In summary, we make the following major contributions:

- We design and implement an approach [1] for effectively generating interface properties from a few generic, high level robustness rules and apply it to the the well known POSIX-API system interfaces.
- We propose a framework [2] 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 explore how we can automatically and scalably infer system-specific intra- and inter-procedural properties using statistical analysis on model checking traces.

4. PRELIMINARY RESULTS

We applied the property generation framework on 10 Redhat 9.0 open source packages. These 10 packages include nearly 100K lines of `C` code. We specified six simple generic properties, all of which pertain to the safe usage of memory pointers that hold the interface return values. Roughly, 1000 concrete formal properties ($>$ 30,000 lines) were generated from 6 generically specified rules ($<$ 60 lines) for 280 POSIX-API interfaces, highlighting the effectiveness of our approach. For static verification, 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. For these 60 APIs, more than 300 concrete rules were generated and they were checked against the 10 Redhat-9.0 open source packages for robustness violations. Given these properties, the static analyzer detected around 200 robustness problems in 10 Redhat-9.0 packages. For static verification, we used a publicly available static analyzer called MOPS [7] with our data flow extensions. Table 1(a) presents the total number of robustness property violations identified by our tool for each of the checked packages. We have shown the API-level violation breakdown for

Table 1: Robustness violations detected for the open source packages

package	# errors
ftp-0.17-17	18
ncompress-4.2.4-33	6
routed-0.17-14	15
rsh-0.17-14	9
sysklogd-1.3.31-3	27
sysstat-4.0.7-3	24
SysVinit-2.84-13	64
tftp-0.32-4	14
traceroute-1.4a12-9	7
zlib-1.1.3-3	4

(a) Overall Errors 10 Packages

API	# errors	API	# errors
fdopen	1	chdir	2
closedir	1	fstat	3
fflush	2	malloc	1
fileno	1	open	2
fputc	1	fclose	12
fputs	2	putchar	1
fseek	2	unlink	4
ftell	1	write	4
getpwuid	1	setuid	1
close	26		

(b) Errors from SysVinit-2.84-13

Table 2: 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

one selected package (`SysVinit-2.84-13`) in Table 1(b). We reported the detailed results elsewhere [1].

For the experiments above, the POSIX specification database for 280 POSIX-APIs was manually built. We used our inference framework to infer specifications for 60 critical APIs selected for our property generation experiments. For these 60 APIs, more than 100 concrete triggers were automatically generated by the `propertyEngine` and they were used against the same 10 Redhat-9.0 open source packages for specification inference. Table 2 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 interace specification could be inferred from the 10 packages we analyzed. An API specification is said to be inferred from the analysis of a set of packages, if it can be inferred by 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 successfully inferred the specifications for 22 APIs. Hence, just by analyzing 10 packages, we inferred specifications for 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. When we analyzed the 10 open source packages using the manually specified `specDB` with the approach [1] shown in Figure 1, we found 188 robustness violations. Using the inference framework proposed on these 10 packages, we could detect 28 out of 188 robustness violations automatically. The false negative ratio decreases as the number of packages analyzed increases. The detailed results were reported elsewhere [2].

5. RELATED WORK

Engler et al. [9] infer bugs by statically identifying inconsistencies from commonly observed behavior. They use simple system-specific static analysis to automatically extract programmer beliefs from the source code, and flag belief contradictions as bugs. They specify a general template for a rule, and allow the automatic analysis to specialize the template to the checked system. We use trigger automata to focus on certain parts of the code (for example, interface $actions$) and infer interface specifications from model-checking traces. In addition, our inferred specifications are combined with generic robustness rules to generate concrete robustness properties for static verification. We also propose a framework for inter-procedural rule inference using model-checking traces.

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 [21] 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. [22] infer 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. Ammons et al. [3, 4] infers formal properties by observing program execution and concisely summarizing the frequent interaction patterns as state machines. 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 verifies inferred properties with static verification. Additionally, test cases are required for generating dynamic traces and they might not exercise all the execution sequences in the source code.

Static compiler analysis has been widely used to find bugs and security holes in source code [14]. The Meta-Compilation (MC) [5] project uses programmer-written compiler extensions to statically find bugs in operating systems and cache protocols. MOPS [7] is a control-flow-sensitive static checker that checks for certain vulnerable system-call sequences in a program. MOPS, however is data-flow-insensitive. Tools like SLAM [6] and BLAST [12] are static analysis tools based on theorem proving and model checking boolean abstractions of a 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 to check. Furthermore, we could adapt these tools for the purposes of property inference. CHET [17] uses model checking techniques to verify properties in moderate-sized Java programs. Our current work focuses on inferring properties from packages written using C. We could adapt the CHET tool for inferring properties from packages written in Java. Liu et al. [15] developed LtRules, which receives a given set of APIs, creates all possible API properties determined by a set of templates, and checks the generated API properties against "good" software packages by using the BLAST [12] model checker. The template instantiations that pass the BLAST test are considered to be properties. These properties are used for verifying other programs. This technique requires "good" reference test programs and fails to infer properties if reference programs have bugs. Our approach generates triggers from user-specified APIs instead of concrete properties like LtRules. As we use statistical techniques for property inference, we can still infer properties from buggy programs.

Dwyer et al. [8] proposed a pattern-based approach that can accommodate properties typically specified with temporal logic or regular expressions. The PROPEL [19] approach provide templates

that explicitly capture details for patterns that commonly occur in the properties that are created for model checking and other types of analysis. With Propel, users are shown the evolving property specification in both natural language sentences and graphical finite-state automata (FSA). Developers can use tools such as PROPEL 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.

Program traces have been previously used for various purposes such as program understanding and debugging. Groce et al. [11] present techniques for modifying a program such that it can produce exactly those executions consistent with a given trace of events, enabling efficient analysis of the reduced program. Howard et al. [13] propose a framework for analyzing traces by checking them against a formal model using a standard model checker. We use a model checker to output program traces related to the interfaces and use them for property inference.

6. CONCLUSIONS

We designed and implemented a novel approach [1] to effectively generate a large number of concrete interface robustness properties for static verification from a few generic, high-level user specified robustness rules for exception handling. Results showed that we can effectively generate more than a thousand useful interface robustness properties for 280 POSIX-APIs from 6 generically specified rules. Given these properties, the static analyzer detected around 200 robustness problems in 10 Redhat-9.0 packages. Interface specifications, required for the generation of concrete properties, and many system-specific properties are often not documented by developers. Our research proposes a novel framework to automatically infer system-specific interface specifications [2] and temporal properties from program source code. We use a model checker to generate static traces related to the interfaces. The interface specifications and temporal properties are inferred from these traces using statistical analysis and certain heuristics. We implemented a prototype for an existing static analyzer with our data-flow and trace-generation extensions. The preliminary results are promising. With our framework, we successfully inferred the specifications for 22 out of 60 POSIX-APIs by analyzing 10 Redhat-9.0 packages.

7. REFERENCES

[1] 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*, pages 293–296, 2006.

[2] M. Acharya, T. Xie, and J. Xu. Mining interface specifications for generating checkable robustness properties. In *Proc. International Symposium on Software Reliability Engineering, To Appear*, 2006.

[3] G. Ammons, R. Bodik, and J. Larus. Mining specifications. In *Proc. Symposium on Principles of Programming Languages*, pages 4–16, 2002.

[4] G. Ammons, D. Mandein, R. Bodik, and J. Larus. Debugging temporal specifications with concept analysis. In *Proc. Programming Language Design and Implementation*, pages 182–195, June 2003.

[5] 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.

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

[7] 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.

[8] 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.

[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] A. Groce and R. Joshi. Exploiting traces in program analysis. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems*, pages 379–393, 2006.

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

[13] Y. Howard, S. Gruner, A. Gravell, C. Ferreira, and J. C. Augusto. Model based trace checking. In *Proc. Workshop on Software Testing*, 2003.

[14] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proc. USENIX Security Symposium*, pages 177–190, 2001.

[15] C. Liu, E. Ye, and D. Richardson. LtRules: an automated software library usage rule extraction tool. In *Proc. International Conference on Software Engineering*, pages 823–826, 2006.

[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] S. Reiss. Specifying and checking component usage. In *Proc. Symposium on Automated Analysis-Driven debugging*, pages 13–22, 2005.

[18] B. Schwarz, H. Chen, D. Wagner, G. Morrison, J. West, J. Lin, and W. Tu. Model checking an entire Linux distribution for security violations. In *Proc. Annual Computer Security Applications Conference*, pages 13–22, 2005.

[19] R. Smith, G. Avrunin, L. Clarke, and L. Osterweil. PROPEL: An approach supporting property elucidation. In *Proc. International Conference on Software Engineering*, pages 11–21, 2002.

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

[21] 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.

[22] 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.