Generating and Inferring Interface Properties for Static Analysis

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

ABSTRACT

Software robustness and security are critical to dependable operations of computer systems. Robustness and security of software systems are governed by various temporal properties. Static verification has been shown to be effective in checking temporal properties. But manually specifying these properties is cumbersome and requires knowledge of the system and source code. Furthermore, many system-specific correctness properties that govern the robust and secure operation of software systems are often not documented by the developers. We design and implement a novel framework 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. These generic rules are free from any system or interface details, which are automatically mined from the source code. We report our experience of applying this framework to test robustness of POSIX-APIs in Redhat-9.0 open source packages. Security properties that dictate the ordering of certain system calls are usually inter-procedural unlike robustness properties. In this paper, we present our ongoing research that infers these properties directly from the program source code by applying statistical analysis on model checking traces. We are implementing our ideas in an existing static analyzer that employs pushdown model checking and the `gcc` compiler.

1. INTRODUCTION

Robustness and security of software systems are governed by various temporal properties. Violations of these properties often lead to system crashes, leakage of sensitive information, and security compromises. This paper focuses on effectively generating these properties for static verification and proposes techniques for automatically inferring these properties from the program source code.

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. The Fuzz project [10, 15, 16] tested the external interface robustness of UNIX utilities by generating random input streams. The Ballista project [12, 13, 19] uses extreme input parameter values to test the robustness of systems call interface implementation. Similar approaches to that of Ballista have also been applied to system call interfaces on Windows [11, 17, 20]. 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 [2–4, 7, 14, 18, 21] them against a software system. However, manually specifying a large number of interface properties for static verification is often inaccurate or incomplete, apart from being cumbersome.

To address these issues, we propose a novel framework that generates interface properties from a few generic, high level robustness rules that capture interface behavior. Generic robustness rules are specified at an abstract level that needs no knowledge of the source code, system or interfaces. These generic rules are then translated by our framework into concrete properties, verifiable by static analyzers. The translation uses the information of the interfaces and system, mined directly from the source code with the help of the `gcc` compiler and certain data flow extensions. We implement our framework for an existing static analyzer with our data flow extensions and apply the framework to the well known POSIX-API system interfaces. We analyze 10 Redhat-9.0 packages that use these system interfaces.

While robustness properties are defined over interfaces, security properties often involve multiple system calls. Most security properties can be defined by certain temporal orderings of system calls. These robustness and security properties are usually system and application specific. Documentation is usually not available for such properties. In this research, we explore how we can automatically infer robustness and security properties using statistical analysis on model checking traces.

The rest of this paper is organized as follows. Section 2 presents our framework for effectively generating interface robustness properties. Section 3 presents our preliminary evaluation results on checking 10 Redhat-9.0 packages against the generated interface robustness properties. Section 4 discusses our proposed techniques to infer robustness and security properties from the program source code. Section 5 concludes the paper.

Proceedings of MASPLAS'06 Mid-Atlantic Student Workshop on Programming Languages and Systems Rutgers University, April 29, 2006

2. ROBUSTNESS PROPERTY GENERATION

This section introduces our framework that effectively generates interface robustness properties for static verification. The goal of our framework is 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 interface or its elements.

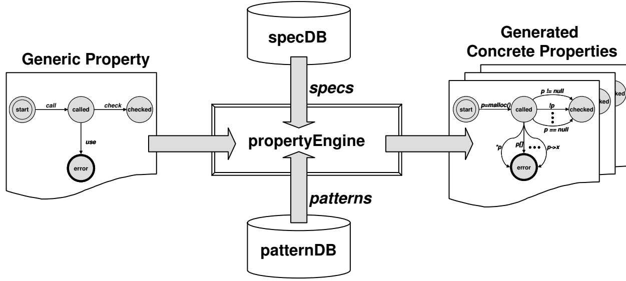


Figure 1: A Framework for Concrete Property Generation

The overview of our framework 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. 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. In the subsections that follow, we identify the elements and actions that characterize an interface and show how generic rules are defined and concrete properties derived from them.

2.1 Interface Characterization

A set of interfaces (such as functions to be invoked) implemented for a specific purpose have similar structural details at a high level. We characterize an interface with its structural elements (such as function parameters or 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

$$spec(i) = \{is(i), rs(i), ss(i), \mathcal{R}, \mathcal{S}, \mathcal{Z}\}$$

$is(i)$ 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, $is($`malloc`$)$ is $\{x\}$, $rs($`malloc`$)$ is $\emptyset$ and $ss($`malloc`$)$ is $\emptyset$. After the statement execution, $rs($`malloc`$)$ is $\{$`p`$\}$ and $ss($`malloc`$)$ is $\{$`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 $($`p`$,$ `mptr`$) \in \mathcal{R}$. If it fails, `p` assumes value `NULL` and $($`p`$,$ `NULL`$) \in \mathcal{R}$. Because the result set and status set are the same for `malloc`, we have $\mathcal{S} = \mathcal{R}$. The set $\mathcal{Z} = \{$`mptr`$,$ `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 $action$ set for the POSIX-API interfaces comprises $alias$, $call$, $check$, $FALSE$, $failure$, $free$, $pass$, $return$, $success$, $TRUE$ and use. The meanings of the actions are self-evident. For example, whenever a return variable is aliased, the action performed on $v \in rs(i)$ (the return value of the interface execution) is $alias$. The action of invoking the interface is $call$. The action of checking the interface return value or status against members in $\mathcal{Z}$ is $check$. If $check$ fails, the action is $FALSE$ (for example, if the check `p==NULL` fails, the action is $FALSE$ and `p` assumes non `NULL` value) The interface execution can either be a $failure$ or $success$. When the return variable is passed to another function, a $pass$ action is said to be performed. When the function in which the interface is executed returns, the action performed is $return$. If $check$ succeeds, the action is $TRUE$. Finally, when the return value is used in program expressions, the action is use. The interface specification does not have any system or language specific details. Details for each of the actions and elements are stored in the `patternDB`. We next show how interface robustness properties can be formed by defining generic rules over interface elements and actions.

2.2 Generic Robustness Rules

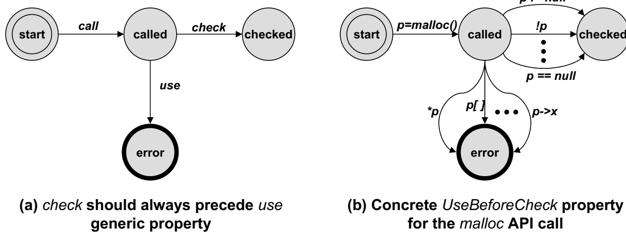
Generic rules for an interface $i \in \mathcal{I}$ are defined over the members of the $action$ set of $\mathcal{I}$. A generic rule is some ordering constraints on the members of the $action$ set. We use the representation given by Dwyer et. al. [6] shown in Figure 2 to represent generic rules. In the figure, A and B are considered to be interface actions.

Property Patterns
- Occurrence: Occurrence of given action during system execution
  - Absence: No A in scope
  - Universality: A throughout scope
  - Existence: A must occur in scope
  - Bounded Existence: A occurs *k* times in scope
- Order: Relative order of actions during system execution
  - Precedence: A must precede B
  - Response: A must follow B
  - Chain Precedence: *m* A's should precede *n* B's
  - Chain Response: *m* A's should follow *n* B's

Figure 2: Action Patterns

Most of the robustness properties of interest can be typically specified by temporal logic or regular expressions. To facilitate property specification and later property checking, we do not consider properties defined by rules such as Chain Precedence that cannot be represented using a regular expression. 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 $check$ action should always be preceded by the use action. The FSM for such a rule is shown in Figure 3(a). Generic robustness rules are currently manually 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

To generate 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 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).

For our preliminary experiments and results, we manually generated the specification database for more than 280 POSIX-APIs. The simplified `spec-DB` for POSIX-APIs is shown in Figure 4. 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 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, correct usage of a given interface or a set of interfaces cannot be immediately inferred from inspecting source code. We propose to build a miner that uses the `gcc` compiler to automatically extract such specifications for the interfaces used in the program. We implement simple data flow analysis in the miner that allows to track the return variable to infer check routines, exceptional return values, etc. The information can be used to build the specification database `specDB` for all the interfaces used in the program. The `pattern-DB` is a constant file specific to each programming language and contains the source code information for different language operations (e.g., dereference, check) that can be performed on simple and complex or dervied data types. The `pattern-DB` can also be built for languages such as C++ and Java.

3. PRELIMINARY RESULTS

We apply our framework to the widely used POSIX-APIs and check the generated concrete robustness properties for Redhat-9.0 packages. We informally enumerate the rules that we used to check various open source packages as below. These properties reflect the most common robustness violations for POSIX-APIs found in

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: Selected Entries from the `specDB` for POSIX-APIs (simplified for presentation)

software packages that use them.

- *UseBeforeCheck*: If the return value of an API call is a memory pointer on success and a `NULL` pointer on failure (we call this API call a *p:np* API call), a variable that holds the return value should always be checked before use or dereferencing.
- *CheckDoesNotExist*: If an API call returns only an integer status value upon success or failure (e.g., `setuid` returns 0 on success and -1 on failure), the status value should be checked for failure.
- *NullPointerDereferencing*: A variable `r` that holds the return value of a *p:np* API call should not be dereferenced on a `NULL` path, which is a path where `r` assumes `NULL` value (e.g., a path to a location inside the true branch of "`if (r == NULL)`").
- *NullPointerFree*: A variable that holds the return value of a *p:np* API call should not be `free`'d on a `NULL` path.
- *FreePointerDereferencing*: A pointer variable that is `free`'d should not be dereferenced.
- *DoubleFree*: A pointer variable is never `free`'d more than once along all execution paths.

Figure 5 shows the composite generic FSM for all *p:np* properties for `malloc`. The "`==NULL`" transition denotes a comparison with `NULL` and the "`!=NULL`" transition is the generic keyword for a comparison with non-`NULL`. The "`==cmp`" state denotes that the return variable is compared with `NULL` and "`!=cmp`" denotes that the return variable is compared with non-`NULL`. The "`==cmp`" state transits to the `NULL` state on TRUE indicating that the return variable is `NULL` on the TRUE path. A $deref$ action (same as use) in the `NULL` state causes a *NullPointerDereferencing* violation. The meanings of other keywords are self-evident. The generic FSM is the composite of five *p:np* robustness property stated above.

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 static verification, we used a publicly available static analyzer called MOPS [4, 5], which employs pushdown model checking to detect 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 [9] 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 the path as the error trace that violates the concrete robustness property.

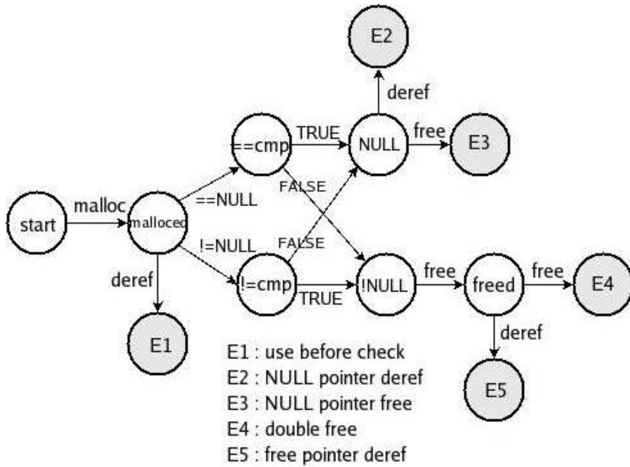


Figure 5: The generic FSM showing all *p:np* properties applied to `malloc`

The generic properties listed above are 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 in order to make it possible to 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 properties (in the form of FSMs) generated by the `propertyEngine` are given to the static analyzer enhanced with our data flow extensions. We evaluate the effectiveness and usefulness of our framework as follows.

Effectiveness: A user only needs to specify a small set of generic properties at a high level. The `propertyEngine` automatically generates more than a thousand concrete properties from 6 generically specified rules for 280 POSIX APIs. 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. We then generated concrete properties for them across 6 generic properties using our property generation framework. These APIs are frequently used in applications and their safe and robust usages are critical for reliability and security. 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.

Usefulness: Table 1(a) presents the total number of robustness property violations our tool found for each of the checked packages. We found around 200 robustness violations in 10 Redhat-9.0 open source packages. We have shown the API-level violation breakdown for one selected package (`SysVinit-2.84-13`) in Table 1(b). Of the 60 analyzed APIs, 19 of them gave violations with this package. We reported the details of our experimental results elsewhere [1]

4. PROPERTY INFERENCE

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

Many system and 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.

Engler et al. [8] show how certain properties can be mined by inferring deviant behaviors in the source code. We use this observation to infer likely rules that a system specific interface or a set of interfaces need to obey. We use the `gcc` compiler and simple data flow analysis to output traces containing generic keywords for a given interface or set of interfaces. Statistical analysis is applied over these traces to infer generic rules, which then can be used by our framework to generate concrete properties. We restrict the trace generation to be intra-procedural because most robustness properties can be captured by such an analysis.

Intra-procedural analysis is sufficient to extract most robustness properties from the 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 use the inter-procedural push down model checking procedure of our static analyzer to generate program traces. Our framework mines security properties from these traces. We use program slicing techniques to reduce the trace size. Program slicing causes the model checker to output only the program statements that are relevant to the set of system calls under consideration. This 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. We are implementing our ideas in the `gcc` compiler and an open source static analyzer that employs push down model checking.

5. CONCLUSIONS

We have showed how large number of interface robustness properties can be generated for static verification from a few generically specified rules. The users need no knowledge of system, source code or interfaces to write generic rules. These details are mined directly from the source code using the `gcc` compiler and certain simple data flow extensions. We implemented this framework for an existing static analyzer with our data flow extensions and applied it to the well known POSIX-API system interfaces. Many system-specific correctness rules govern the robust and secure operations of software systems; but these rules are often not documented by the developers. We presented our research ideas for inferring intra-

procedural generic robustness rules and inter-procedural security properties directly from the program source code by applying statistical analysis on model checking traces. We are implementing our ideas in the `gcc` compiler and an open source static analyzer that employs push down model checking.

6. REFERENCES

[1] M. Acharya, T. Sharma, J. Xu, and T. Xie. Effective generation of interface robustness properties for static analysis. In *submission to the International Conference on Automated Software Engineering (ASE)*, 2006.

[2] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of IEEE Symposium on Security and Privacy*, 2002.

[3] T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of Workshop on Model Checking Software, SPIN*, 2001.

[4] H. Chen, D. Dean, and D. Wagner. Model checking one million lines of C code. In *Proceedings of NDSS 2004*, February 2004.

[5] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *Proceedings of CCS 2002*, November 2002.

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

[7] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design (OSDI)*, 2000.

[8] 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 *Proceedings of 18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[9] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking push down systems. In *Proceedings of CAV*, 2000.

[10] J. Forrester and B. P. Miller. An empirical study of the robustness of Windows NT applications using random testing. In *Proceedings of 4th USENIX Windows Systems Symposium*, August 2000.

[11] J. Haddox, G. Kapfhammer, C. Michael, and M. Schatz. Testing commercial-off-the-shelf software components. In *Proceedings of the 18th International Conference and Exposition on Testing*, 2001.

[12] P. Koopman and J. DeVale. The exception handling effectiveness of posix operating systems. *IEEE Transactions on Software Engineering*, 26(9), September 2000.

[13] N. Kropp, P. J. Koopman, and D. P. Siewiorek. Automated robustness testing of off-the-shelf software components. In *Proceedings of IEEE International Symposium on Fault-Tolerant Computing (FTCS)*, 1998.

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

[15] B. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12), December 1990.

[16] B. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. *Computer Science Technical Report 1268, University of Wisconsin-Madison*, 1995.

[17] M. Schmid, A. Ghosh, and F. Hill. Techniques for evaluating the robustness of Windows NT software. In *Proceedings of the 2000 DARPA Information Survivability Conference and Exposition (DISCEX'00)*, January 2000.

[18] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of 10th USENIX Security Symposium*, 2001.

[19] C. Shelton, P. Koopman, and K. DeVale. Robustness testing of the Microsoft Win32 API. In *Proceedings of IEEE International Conference on Dependable Systems and Networks (DSN)*, June 2000.

[20] T. Tsai and N. Singh. Reliability testing of applications on Windows NT. In *Proceedings of IEEE International Conference on Dependable Systems and Networks (DSN)*, June 2000.

[21] D. Wagner, J. S. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of Network and Distributed System Security Symposium*, February 2000.