1 Introduction

Most of the application testing focuses on the interface between the user and the application. However, software applications do not only interact with their users but also with resources and remote entities. Some recent approaches focus on testing the interface between the application code and the underlying software components like shared libraries. For example, one interesting approach is error injection [1, 3] where possible error return values of library functions are injected at the interface between the application code and the shared libraries.

Error injection simulates failures of the environment of an application. To do so, an error injector intercepts function calls from the application to the library. When injecting an error, the wrapper returns an error value of the called function \( f \) instead of executing \( f \). This is normally done by returning a certain error value (e.g., NULL or \(-1\) depending on the specification of the library function) or throwing an exception. Error injection tools like FIG [1] and AutoPatch [3] need some form of specification of the possible error return values. A correct error specification is complete, i.e., contains all possible error return values and sound, i.e., does not contain error return values that the function does not return. Defining a correct error specification for all shared libraries is a large-scale effort. First, the specifications of libraries are often informal, incomplete, incorrect or just does not exist. Hence, the error specification of a library has to be extracted manually from the documentation or better from the source code because documentation is often out-of-date or contains mistakes. Second, typically there is a large number of libraries installed on a system. If one wants to do a comprehensive error injection, it is not sufficient to extract the error specification from the most common libraries only.

We propose a novel technique to learn error signals of library functions by injecting errors on the system call level. Therefore, we intercept syscalls issued by libraries and return an error instead of executing the syscall. When the library passed the control back to the application, we record the error return value. This approach has several advantages. First, the syscall API is small compared to the number of function implemented in all libraries installed on a common system. Second, the syscall API is well defined and sufficiently stable. Hence, a small hand-written error specification of the syscall API is sufficient to derive sound error specification for library functions that use syscalls directly or indirectly. At the end of this paper we show how to mine the error specification of the Linux syscall API automatically. Third, an error specification of a library can be reused for injecting errors in other applications that use this library.

To derive an error specification of a library, we execute applications that use this library under the control of our syscall error injector. All errors returned by the libraries are recorded, in particular, the function’s return value and value of the errno variable. The recorded error values form a sound but possibly incomplete error specification. We use this error specification later on to inject errors into an application when performing library calls. This is depicted in figure 1: to get the error specification needed for injecting errors from a library into an application, we inject errors into the library at the syscall level.

Please note that always injecting errors at syscall level has some disadvantages compared to our approach. First, it would be more difficult to infer where the actual bug is located. Injecting errors at the syscall level will expose software bugs at the library and application level while our approach will in general only trigger application bugs.
ond, we can combined the injection results of multiple applications and in this way will be able to increase the number of different errors injected at the library level.

2 Details

Our implementation consists of two components. First, a wrapper that intercepts functions calls from the application to the library. Second, an interceptor that intercepts syscalls made by libraries calls. The interceptor can return an error instead of performing the syscall when the call is performed by a library function. To be able to distinguish between syscalls from a library and the application, the library wrapper tracks calls to library functions. It also records the return values and the errno set by library functions.

This approach can be extended to other programming languages and error handling paradigms. For example, for Java the wrapper would intercept calls to the methods of certain classes. It informs the syscall error injector that a library function is going to be executed and then calls the wrapped method. All exceptions thrown by the wrapped method are caught and recorded. This exceptions form the error specification of the executed method. Note that the exceptions that a method could throw are part of the method’s signature in Java. However, runtime exceptions are excluded from this rule. They would nevertheless become part of our error specification.

Two issues are left to be discussed. First, efficiency of the error injection and second, we show how the error specification of an operating system API can be obtained.

Efficient Error Injection Just before injecting an error, we take a checkpoint of the application’s and the library’s state by forking a new child process. The error injection is done in the child process. The child process is only executed until the error signals of the library function has been recorded by the wrapper. The parent process waits until the child process has finished and continues with normal execution until the next syscall is encountered. This approach is similar to that of [2].

Without using forks the application would have to be executed once for each error that we want to inject. If \( N \) is the number of syscalls performed by library functions during a certain run of an application, then our analysis would need to execute \( O(N^2) \) syscalls. With the fork approach the number of syscalls reduces to \( O(N) \). However, the forking of a child process introduces a new issue.

We must allow the child process to perform syscalls after the error injection (but before the library’s error signal can be recorded). This might however corrupt the external state of the analyzed application. For example, consider an application that writes to a file. If the client process writes to this file in its short life time, than the file might be corrupted. We are currently examine techniques to rollback the changes that the client has done. Unfortunately, that might not always be possible. For instance, a send operation to a remote site can not be rolled back. Another alternative that we are currently exploring is to not execute any syscalls after injecting an error. However, this might introduce some internal consistency issues because the external state does not change in a way the code might expect.

Obtaining the OS Error Specification For the error injection on the syscall level we need an error specification of the syscall API. While this API is significantly smaller than the number of library functions, one might still want to create this error specification automatically. We present an approach that infers this error specification for Linux with the help of kernel header files and the syscall documentation. First, we parse the kernel header file where the syscall numbers are defined (unistd.h). This results in a mapping from numerical syscall ids to textual syscall ids. Second, we parse the kernel header file where all error codes of all syscall operations are defined (errno.h). Again, the result is a mapping from numerical errornos to textual error names. Third, we parse the man pages for each syscall for all textual error names from the errno-mapping. We assume that all error names mentioned in the documentation of a syscall can be returned by it.

3 Conclusion

We presented an approach to learn the error return values of library functions. The main idea is to first inject errors at the syscall level (while a library function is executed) and second, to record all error values returned by a library function. The error specifications are useful for injecting errors from libraries into applications. We expect that an error specification obtained from a few runs of a small set of applications can be used to inject errors in a large set of applications.

References