Automatically Finding and Patching Bad Error Handling

Martin Süßkraut  Christof Fetzer
Department of Computer Science
Technische Universität Dresden, Germany
{martin.suesskraut,christof.fetzer}@tu-dresden.de

Abstract

Bad error handling is the cause of many service outages. We address this problem by a novel approach to detect and patch bad error handling automatically. Our approach uses error injection to detect bad error handling and static analysis of binary code to determine which type of patch can be instantiated. We describe several measurements regarding the effectiveness of our approach to detect and patch bad error handling in several open source programs.

1 Introduction

Many service outages are caused by buggy error handling code [2]: the error handling is often the least tested, least documented, least executed and least understood part of a software component. Under high load more resources (e.g., more memory and file descriptors) are needed and errors will occur and will need to be handled correctly when resources become depleted. Bad handling of resource depletion errors are therefore more likely to become visible at times when the system is needed the most, i.e., when the load is high.

Because of economical reasons, most dependable systems cannot be built from scratch. Instead one has to reuse existing software components. Some of these components might however not been built to the standards required for dependable systems. One can use different static and dynamic analysis techniques to help the developers in the process of evaluating and improving the dependability of software components.

Since error handling code can be a large part of a code base ([2] reports up to two thirds) and can be the cause of a large percentage of outages ([2] also reports up to two thirds even though error handling code is rarely executed), we focus in this work on the problem of how to evaluate and improve error handling code. This work is part of the AutoPatch project which tries to increase the dependability of software-based systems by (1) using static and dynamic program analysis to evaluate source and binary code and (2) generating patches for certain robustness issues found in the analysis phase.

The generated patches can be used in different ways. First, they can be used until manual patches become available. This is, for example, important for critical bugs that can be used in security exploits. Second, they can be used to improve the efficiency of developers by helping them to correct their code with less effort. Third, for less critical components they might be used instead of patches supplied by a vendor. This is especially useful if a vendor is not willing to provide patches.

The approach of this work is to use error injection techniques to discover bugs in the error handling of programs. We will present some of the bugs we found in this way in Section 4.2. The basic observation on which our patch generation is based upon is the following: even though error handling is the buggiest part of the code, nevertheless most programs handle most of the errors correctly. We use this observation by trying to map errors that a program cannot handle to errors that the program can handle. To do so, we define patch patterns that can be applied in well specified situations. Whether a pattern can be applied in a given situation is verified using static analysis of the binary code.

The outline of this paper is as follows. We first present related work in Section 2, describe our approach in Section 3 and evaluate it in Section 4.

2 Related Work

This paper is related to other works in the area of robustness analysis and patch generation. The Ballista project [7, 8] provides a toolkit to automatically determine the robustness of POSIX functions using error injection. The functions are called with extreme values as arguments to determine their robustness. Ballista differs in two major aspects from our work: (1) the error injection is done into POSIX functions using extreme values while we inject errors into applications and (2) their results are used for measurements only while we use it for patch generation.
The HEALERS project [5, 4] uses a Ballista style error injection approach to determine safe and unsafe data types for function arguments. Values which belong to unsafe data types lead to non-robust behavior (e.g., crashes) or insecure behavior (e.g., buffer overflows). These data types are used to generate various types of wrappers for shared libraries. For example, a robustness wrapper prevents a library function from being invoked with unsafe argument values. Similar to the wrappers generated by HEALERS, our patches intercept function calls. However, in this work we protect the application and not the library as it was done in HEALERS. Both approaches are complementary because healers makes sure that libraries return errors instead of exhibiting non-robust or insecure behavior and our approach makes sure that programs can cope with errors returned by libraries.

We reuse some of the techniques of HEALERS to determine the type of shared library functions: the Unix _man_ pages and C header files are parsed to extract function signatures. HEALERS uses these signatures for selecting the extreme values to test a function. We use the signatures for static analysis and for patch generation.

FIG [1] is a project to determine the robustness of applications in the presence of errors returned by system functions. It does so by error injection from the interface of the Standard C library into the application. To do so, wrappers are generated that intercept calls to functions of the Standard C library. Depending on the given configuration, either an error value is returned or the wrapped function is executed. The authors of [1] have found several robustness issues in applications. Like Ballista, the error injection is done to evaluate the robustness only. We use a similar approach as FIG for injecting errors. We use this to identify calls to library functions for which an application does not perform proper error handling. We use this information not only for evaluation purposes but also to determine which function calls need to be patched.

The authors of [10] automatically generate security patches for applications. Their approach tries to patch the vulnerability abused by an exploit. Additionally the application’s source code is needed. The application’s source code is compiled with a library to detect buffer overflow attacks. The application is executed and the exploit is performed on it. If a buffer overflow is detected, the data of the buffer overflow protection library is used to alter the applications source code to prevent this buffer overflow. The authors introduce a common patch pattern to alter the application’s source code. Although the patches are also automatically generated our work differs in its requirements and focus. We focus on robustness issues. As we use error injection, we do not need any knowledge about robustness flaws of the inspected application. Also, we do not need the application’s source code. Our patches are wrappers located between an application and its libraries. Using wrappers we do not need to alter the application’s binary or source code.

3 Automatic Finding and Patching of Bad Error Handling

Systems-oriented libraries like the standard C library return error codes when a failure occurs during the execution of a function. For example, if all file descriptors are currently in use, an error code is returned by functions that open a file. Dependable applications need to handle such error codes appropriately. Error handling code is, however, often buggy. Such bad error handling can be a major cause for service unavailability.

In this work we focus on bad handling of errors returned by systems-oriented libraries. We use systematic error injection to identify function calls that do not handle errors properly. We call this technique error injection. We understand that under proper handling of errors an application does not crash when we inject an error code. In this work we do not address other possible consequences of bad error handling like an incorrect output or application hangs.

To perform error injection into applications on the library level, we intercept calls to library functions and return in some cases an error code instead of executing the function that was called. We do this in a systematic way to identify unsafe function calls. Function calls are unsafe if the application crashes when an error value is returned. These are the function calls that must be patched. Our analysis is restricted to calls to library functions, which we can and want to patch. We use the term function of interest to refer to such library functions.

Patching of the application is based on patch patterns. In this paper we present two patch patterns: (1) masking the failure of the called function by Preallocation and (2) by using the application’s error handling of function calls that do not result in crash by Error Value Mapping. The basic idea of the preallocation pattern is that one maintains some reserved resources for calls that reserve resources and for which the application cannot cope with errors. The error value mapping pattern transforms an error of a call to a function that the application cannot deal with, into an error that is returned to some other call for which the application can deal with errors. Both patch patterns cannot guarantee 100% robustness for the patched bugs. But our evaluation shows that the patches reduce the number of crashes significantly.

Additional information about calls to library functions is needed in order to apply our patch patterns. The preallocation pattern needs to know the argument values of these function calls. The values are used as parameters for preallocation, which must be done before the unsafe function call. We use a dynamic and a static approach to obtain them: (1) we record dynamically all arguments for function calls
of a specific run of the application and (2) we do a static analysis of the application’s binary to extract further information that helps us to generate the patches. Figure 1 shows the data flow. The argument recorder and the error injector are dynamic analysis techniques – they execute the application and observe it. The user must provide a run configuration (work directory, command line arguments, etc.) for repeatable execution. The static analysis tool works on the application’s binary. So our approach does not depend on the application’s source code. If the binary contains debugging information, our system can also extract various information that will aid a developer in fixing the bugs the system finds.

All data gathered by our analysis tools is stored within a database. There are two reasons for using a database: (1) some of the data gathered is \textit{expensive} to acquire and one wants to keep it persistent. For instance, the error injection tool has to run the application for each function call it performs. Hence, instead of performing the \( N \) function calls of that application, \( N^2 \) functions calls are performed by the error injection tool. (2) The error injector and our static analysis need some of the data gathered during argument recording. Therefore, it is very convenient to have the data accessible via the database.

Please note that our approach does not rely on already found bugs. The only inputs our approach needs are the application’s binary and one or more run configurations of the application. For example, existing unit tests of the application could be used as a run configuration. The error injector needs the function names and the return values that signal an error. We call these values \textit{error values}. These have to be given by an expert unless this cannot be derived automatically, e.g., by parsing the man pages. The signatures of functions used within the various generated wrappers are derived with the approach of \cite{5}.

In the following we describe our analysis approaches: (1) the argument recorder, (2) the systematic error injector and (3) the static analysis tool. This section concludes with the details of the patch generation and the two patch patterns \textit{Error Value Mapping} and \textit{Preallocation}.

### 3.1 Argument Recording

The objective of the argument recorder is to learn common arguments for specific invocations of functions of interest. For example, the preallocation pattern tries to keep some preallocated resources as a backup. Knowing the typical resource demands for a call, we are able to optimize the size of the preallocated resources.

The argument recorder learns the return addresses of calls to functions too. The return addresses within the application’s binary are used as an identification of function calls. The advantages of return address as identification of function calls are as follows. First, the address is independent from a certain execution and the scheduling behavior of multithreaded applications. It is also used to tie the dynamic analysis to the static analysis: the return address of a function call can be extracted from the disassembled code (see Section 3.3).

In summary, our argument recorder works as follows: (1) a wrapper is generated that intercepts all invocations of functions of interest to record their argument values, (2) the application is executed with this wrapper preloaded, and (3) the recorded argument values are written to the database for the next analysis steps and patch generation.

Our argument recorder is a dynamic analysis tool and hence, the recorded arguments are associated with specific runs of the application. There are two issues with this approach. First, there is a coverage issue: a call to a function of interest might not be executed in the run we used for recording the arguments. Second, the argument values of a function call might be specific to a run and not a good estimate of a typical argument value. Our approach is to run the application multiple times in different configurations within the argument recorder. The results of all runs of an application are included in all further analysis and patch generation.

In the following we describe the working of the argument recorder in more detail. Before running the application, an argument recording wrapper is generated. The purpose of this wrapper is to intercept all calls to functions of interest and to print the argument values and return address of
all intercepted calls to our database. The return address of the current function can be found on the stack. A wrapper function writes the current argument values and the current return address to the database and then it calls the function it wraps. The following pseudo code illustrate the operation of an argument recorder wrapper:

```c
return_type function_name(type1 arg1, type2 arg2, ...) {
    write_to_db(return_address, arg1, arg2, ...);
    return original_function(arg1, arg2, ...);
}
```

The argument recorder gets a run configuration from the user. This configuration is saved to the database to reexecute the run within the error injector (see Section 3.2). The run configuration consists of: the applications path, command line arguments, the working directory of this run and optionally some setup script. The setup prepares the external state of the application (for example it can be used to remove files left over from a previous run). The argument recorder sets up the environment to run the application by running the setup script and changing the working directory. The application is run with the configured command line arguments. The argument recorder preloads the wrapper with the main function name. The argument recorder then calls the function.

The analysis works as follows: for each function call to a function of interest grouped by return addresses an error injection wrapper is generated. The wrapper intercepts the call to the function of interest. It checks if the current return address matches the return address of the wrapper. If this is true, it returns a given error value, otherwise it returns the result of the invocation of the wrapped function. The following pseudo code illustrates the operation of an error injection wrapper:

```c
return_type function_name(type1 arg1, type2 arg2, ...) {
    if (current_return_address == error_injection_return_address)
        return error_value;
    return original_function(arg1, arg2, ...);
}
```

For each function call the application is run with the corresponding wrapper preloaded to test this function. The same run configuration as for argument recording is used by the error injector. The error injector uses the same preloading approach as the argument recorder. After running the application, the error injector waits for the application to terminate. It records, if the application crashes or exits and starts the application with the next wrapper.

The approach makes four assumptions: (1) that the system behaves good, i.e., does not signal errors: our wrappers are the only parts of the system that return error values and the application will not crash because of other errors. Note that this assumption can be checked by the wrapper during runtime and executions in which this assumption is violated can be retried. (2) The system has to know the error return value of functions of interest a priori. This can be given by an expert or manually extracted from the specification or documentation of the functions of interest. Automated approaches are also possible. (3) the only bad behavior is a crash of the application. If this is sufficient depends on the application. For instance it might be useful to check the application output for consistency and correctness. But both properties depend on the application. It would be unproblematic to aid the classification by a correctness checking script provided by an expert of the application. We omitted this step to make our approach more generally applicable. (4) Our approach misses bugs, which are only triggered if more than one function call signals errors. The systematic error injector can be extended to search for such bugs by testing all combination of function calls. While this will be simple to implement, the number of wrappers to generate and the runs to execute will increase exponentially with the number of calls to functions of interest. That is why we decided to assume that all bugs are triggered by exactly one unsafe function call. As the error injection is a dynamic analysis technique it suffers from the same code coverage drawbacks as argument recording.

To aid the developer with locating and fixing the bugs found, our error injector wrapper catches all signals sent to the application. In this way, we can provide the developer with the address where the error is injected and the address of the actual crash. If the applications binary contains debugging information, these addresses are translated into a source code file name and a line number. We have used this information to find the bugs presented in Section 4.2. Note that the requirement for debugging information is optional and is not needed for generating the patches.

### 3.2 Systematic Error Injection

Our implementation uses systematic error injection to classify function calls as safe or as unsafe. A function call is safe, if the application does not crash when the function called returns an error value. All function calls for which the application crashes if they return an error value are unsafe. Even if a crash was only observed for one invocation in one run and all other observed invocations of this function call do not crash we treat the function call as unsafe. Unsafe function calls have to be patched. The results of the systematic error injection is written to the database after the classification.

The analysis works as follows: for each function call to a function of interest grouped by return addresses an error injection wrapper is generated. The wrapper intercepts the call to the function of interest. It checks if the current return address matches the return address of the wrapper. If this is true, it returns a given error value, otherwise it returns the result of the invocation of the wrapped function. The following pseudo code illustrates the operation of an error injection wrapper:

```c
return_type function_name(type1 arg1, type2 arg2, ...) {
    if (current_return_address == error_injection_return_address)
        return error_value;
    return original_function(arg1, arg2, ...);
}
```

For each function call the application is run with the corresponding wrapper preloaded to test this function. The same run configuration as for argument recording is used by the error injector. The error injector uses the same preloading approach as the argument recorder. After running the application, the error injector waits for the application to terminate. It records, if the application crashes or exits and starts the application with the next wrapper.

The approach makes four assumptions: (1) that the system behaves good, i.e., does not signal errors: our wrappers are the only parts of the system that return error values and the application will not crash because of other errors. Note that this assumption can be checked by the wrapper during runtime and executions in which this assumption is violated can be retried. (2) The system has to know the error return value of functions of interest a priori. This can be given by an expert or manually extracted from the specification or documentation of the functions of interest. Automated approaches are also possible. (3) the only bad behavior is a crash of the application. If this is sufficient depends on the application. For instance it might be useful to check the application output for consistency and correctness. But both properties depend on the application. It would be unproblematic to aid the classification by a correctness checking script provided by an expert of the application. We omitted this step to make our approach more generally applicable. (4) Our approach misses bugs, which are only triggered if more than one function call signals errors. The systematic error injector can be extended to search for such bugs by testing all combination of function calls. While this will be simple to implement, the number of wrappers to generate and the runs to execute will increase exponentially with the number of calls to functions of interest. That is why we decided to assume that all bugs are triggered by exactly one unsafe function call. As the error injection is a dynamic analysis technique it suffers from the same code coverage drawbacks as argument recording.

To aid the developer with locating and fixing the bugs found, our error injector wrapper catches all signals sent to the application. In this way, we can provide the developer with the address where the error is injected and the address of the actual crash. If the applications binary contains debugging information, these addresses are translated into a source code file name and a line number. We have used this information to find the bugs presented in Section 4.2. Note that the requirement for debugging information is optional and is not needed for generating the patches.

### 3.3 Static Analysis

We do static analysis to aid the patch generation. The patch pattern, we are presenting in Section 3.4, need additional information that is not provided by our dynamic analysis techniques. First, the error value mapping pattern needs to know, if an unsafe function call is in a so called call
group together with a safe function call. A call group is a sequence of function calls that is executed completely or not at all. This implies that there exists no jump statement and no jump target between the corresponding call statements in the application’s binary code. This pattern is used to map a failed call from an unsafe to a safe function call. Second, for preallocating resources the patch generator needs to know the arguments of the function call before the call is executed by the application. One way to get these arguments is to look into the binary code if the argument values for a specific function call are hardcoded into it. If a function call has hardcoded argument values, we say that it has static arguments. If we cannot be sure if a function call is always executed with the same argument values, we say that it has dynamic arguments.

Our implementation has a static analyzer to obtain call groups and static arguments from the application’s binary code. To be independent from the access to the source code of the application, the static analyzer works on the disassembled binary code. Because disassembling a binary is in general a surprisingly difficult problem, we cannot always guarantee that we can perform a static analysis on all the code. Also, we check the results of the static analysis by comparing it with the data obtained by the argument recorder. The static analysis only considers function calls of which function names and return addresses (the address of the next statement after the call statement) have been found by the argument recorder. Nevertheless, there is a possibility for both false positives (false call groups or function calls with false static arguments) and false negatives (where we miss a call group or a function call with static arguments). We discuss these issues below.

We use the disassembly output by the GNU tool objdump [6]. Looking for function calls is simple pattern matching: Whenever a call operation is found, its operands are inspected. If they point to a function of interest, the call itself and its return address (the address of its successor statement in the disassembly output) is stored. Functions of interest can be easily recognized because their names must be part of the binary. objdump includes the function names for calls to external functions into the disassembled operands of call statements.

### Static Arguments

To identify calls with static arguments, we iterate over the predecessor statements of a function call. Because the Linux IA32 ABI [9] defines that argument values to a function call have to be passed over the stack, we are looking for stack operations. We are looking for two possible stack operations:

1. `push <const>` It pushes a constant onto the stack.
2. `mov <const>, <offset>(<stack ptr>)` It moves a constant onto the stack. The stack pointer is always `%esp` and the offset is an integer constant (or zero if omitted).

The number of arguments of a function is known a priori from its signature. So the static analyzer knows the number of statements to inspect (plus the set of offsets, if the `mov` operation is used). Only if all arguments of a certain function call are hardcoded into the binary with `push` or `mov` operations, we classify the arguments of this call as static. The arguments are extracted and saved into the database together with the name of the called function and the return address of the call.

### Call Groups

We show a source code example of a call group in Section 4.2. To identify call groups, we treat all function calls within the disassembly as one single call group. We then split them at the start of functions, so that all function calls within one function are part of the same call group. Next we identify all jumps and jump targets within a function and split the call groups further so that they do not contain a jump or a jump target. We only store call groups with calls to functions of interest into the database.

One prominent source of false positives (misinterpreted call groups and function calls with static arguments) are indirect jumps. The target of an indirect jump is calculated during runtime. In general it is impossible to derive the exact set of possible targets for an indirect jump from the disassembly. Our current implementation skips them. But it would be possible to add heuristics to determine some indirect jump targets. The missed jump targets might lead to too large call groups, when such an indirect jump goes into a call group. It is also possible that such a jump target is between a function call and the statements that move its operands onto the stack. In this case our implementation misclassifies a function call to have static arguments.

We did a survey with a set of sample C programs that contained common syntax constructs. They are compiled with different optimization options and we inspected the disassembly output of the programs. We found no indirect jump as described above. Of course, this is only an indication that for the programs and compiler used indirect jumps are not an issue.

False negatives are also possible. One cause for false negatives that we observed in our survey is tail recursion. The GNU compiler gcc sometimes generates for function calls at the end of a function these statements:

```
leave
jump <function>
```

Instead of calling the function and returning to the current function’s callee, the stack is restored and a `jump` to the function is executed. The return address of the current function is now the return address of the function jumped to. Because we cannot predict the functions return address in general we skip such function calls.


3.4 Patch Generation

The patches we generate are wrappers. Such a wrapper is generated for each application with unsafe function calls. This wrapper intercepts unsafe and safe function calls. Each wrapper is a sequence of instantiations of patch patterns. We have implemented a patch generator for each patch pattern. A micro generator architecture [4] integrates the code produced by these generators into one single patch wrapper per application.

We use two patch patterns to fix unsafe function calls. (1) Preallocation tries to ensure that the resources requested by an unsafe function call are available when the call is executed. (2) Error value mapping maps the error value from unsafe function calls to safe function calls. The assumption is that the unsafe function call can be handled indirectly by the safe call. We have observed that both patch pattern work but they can of course not mask every failure occurring at an unsafe function call.

To test these patterns, we implemented them for calls to malloc, calloc, and realloc. Because the patterns are not restricted to these functions, we describe them in a more general form.

Error Value Mapping We apply the error value mapping patch pattern to call groups with at least one unsafe function call and one safe function call. In short, whenever one of the unsafe function calls within a call group returns an error value, our patch wrapper ensures that the safe function call also returns an error value. Our hope is that instead of crashing, the application handles the error of the unsafe function call together with the safe function call. In Section 4.2 we present a bug found in grep where this pattern is applicable.

Figure 2 shows how errors of unsafe function calls are mapped to a safe call. Function calls f1 to f6 belong to the same call group. If a call group contains more than one safe function call, the last one is chosen to map errors to. We call this safe function call error target. Failures of unsafe function calls with dynamic arguments executed after the error target cannot be mapped back to the error target. That is because the argument values of such calls are not known when executing the error target. The patch works as follows: all return values of unsafe function calls before the error target are stored within the patch wrapper. When the error target call is executed the stored return values of all unsafe function calls executed before the error target are checked. If one of them contains an error value, the error target call returns also an error value. We call this forward mapping because errors are mapped forward to the error target. Errors of unsafe function calls with static arguments executed after the error target are mapped with backward mapping: within the intercepted error target call all unsafe function calls after the error target are performed. The results are stored within the patch wrapper. If one of the results contains an error value, the error target call returns an error value, too. When the backward mapped function calls are performed by the application, the function calls are intercepted and the stored return values are returned. Unsafe function calls with dynamic arguments (like f5 in Figure 2) cannot be patched because their argument values are unknown at time the error target is executed.

The pattern is applicable if the order in which the function calls are issued does not matter. In the call group in Figure 2, the function call f6 is performed together with f4 and before f5. If the order of the function calls must be preserved, no backward mapping is possible. Please note that all calls in a call group can be calling different functions with different error codes. Because the failed unsafe function might differ from the error target, the application’s user might see a wrong error message. To test the influence of the call order one can generate a special wrapper that shuffles the order of calls in a given function group. If the application’s results of executions with this wrapper do not differ from executions without this wrapper, it indicates that the call order does not matter.

We have also implemented a weaker form of error value mapping: the patch wrapper can optionally return error values for all safe function calls as soon as one unsafe function call fails. When the patch detects an error return value for an unsafe function call, it handles it and sets an internal flag. All safe function calls are intercepted and the internal flag is checked. If it is set, an error value is returned. The assumption is, that the application handles the error of the safe function call together with the unsafe one before it crashes. Usually the application will exit returning an error message to the user. That’s why we call this early-exit.

Preallocation The assumption of the preallocation pattern is that one should try to reserve all resources as soon as possible. This approach can be combined with early-exit
to try to gracefully signal a preallocation error to the application through a safe function call. Preallocation for unsafe function calls with static arguments can be done without any additional action: at the start of the patch wrapper, it preallocates the resources for all unsafe function calls with static arguments. Whenever an unsafe call fails, the preallocated resources are used and a flag is set. Whenever a safe function call succeeds, all preallocated resources are checked if they need to be renewed. To renew a resource, it is preallocated. There are two options if the renewing fails: (1) use early-exit and signal an error value at the safe function call that does the renewing. Or (2) ignore the failure. Ignoring early-exit and signal an error value at the safe function call cated. There are two options if the renewing fails: (1) use

to get a set of parameters for preallocation. For example, we take the maximum argument value seen as size parameter for malloc. Other options are to take the average or the average plus the deviation. We did not see any different results for these options in our experiments (see Section 4). When an unsafe function call with dynamic arguments is performed and fails, our patch wrapper intercepts the call. It compares the current argument values with the parameters used for preallocation. If they do not match, some corrective actions can be done (e.g., trying to resize a preallocated block).

The preallocation pattern can only be used for unsafe calls to functions which reserve some resources. So it is much more limited in its use than the error mapping pattern. Preallocation does not preserve the order of the function calls. In case of unsafe calls with dynamic arguments, corrective actions must be possible. All these assumptions hold for the C API for memory allocation: malloc, calloc, realloc, and free. Whenever the size of a preallocated memory chunk does not fit the current argument values, realloc can be used to correct the size of the allocated memory. Other functions for which preallocation can be applied are for instance: opening files (e.g. fopen) and sockets (e.g. socket, bind, connect). We plan to address these functions in future work.

Patch Generation The patch generator sorts all unsafe function calls learned from error injection before generating the patches. We have implemented a micro generator per patch pattern. First all unsafe function calls that can be patched by the call group pattern are assigned to the call group patch generator. Second, all unsafe calls to memory allocating functions with static arguments which are left are assigned to the static preallocation generator. Finally, all unsafe function calls with dynamic arguments to memory allocation functions which are left are assigned to the dynamic preallocation wrapper. Our architecture integrates all generated patches into a joint patch wrapper.

4 Evaluation

We have implemented our approach on top of Ubuntu Linux 5.10. Our four tools are implemented in Ruby 1.8 and we use Postgresql 8.0 as database back end. All wrappers are generated in C. Our implementation is currently restricted to IA32. This is mostly because of the static analysis tool. At least the regular expressions used by the tool would have to be reimplemented for other platforms. We have applied our approach to 10 command line applications that are part of the standard installation of Ubuntu Linux. Each of the 10 applications crashed at least once during error injection. Our generated patches are able to cope with up to 84% of the unsafe function calls we have found. The worst run-time overhead of a patched application was 9.14%.

We will first present the number of unsafe function calls we have found for each of the 10 applications. After that, we will discuss the effectiveness of the generated patches with the help of various error injectors. We will present measurements regarding the overhead of the patches. Finally, we will describe some concrete bugs that our tools have found.

4.1 Measurements

Unsafe Function Calls Figure 3 shows the number of unsafe calls per application and how many of that calls have static arguments. Thus our patches are currently limited to the function of the C memory managing API (malloc, calloc, and realloc) we have only simulated errors of these functions. All applications suffer from at least one unsafe function call. Except for grep and du, the unsafe function calls are performed by the Standard C Library. The crashes are also within the library. For sum, uname, wc, df, md5sum, sort, and touch, the crash happens while executing Standard C Library function setlocale. Application unzip crashed while executing Standard C Library function tzset. Since the unsafe function calls are part of the Standard C Library, they are not taken into account within the static analysis. So our tools assume that these calls have dynamic arguments. In grep 4 of the 6 unsafe function calls have static arguments and in du 2 of 4 unsafe function calls have static arguments.
Robustness Evaluation of Patches Each patched application was first tested without error injection. All runs had the expected output, so none of the generated patches did any harm in the absence of failures in our test setup. To stress the patches, we run various error injection experiments with the unpatched and patched applications. The results for patches without early-exit are shown in Figure 4. We used four different error injectors to stress the generated patches. The first one is the systematic error injector. The second one is called knockover error injector. It works like the systematic error injector (one run per function call) but after it returns an error value for the first time, it will return an error value for all following calls of the current run. This stresses the error handling code. Except for grep and unzip, we have found more crashes with the knockover error injector.

The third error injector maintains a maximum amount of memory. The amount of currently reserved memory is maintained within the error injector. Whenever an application tries to exceed this amount, an error value is returned. We experimented with amounts of 100, 1,000, 10,000, 100,000, 1,000,000, and 10,000,000 bytes. The most crashes were observed for the limit of 1,000 bytes. All of these crashes happened within the C-library at the same return address before the actual application code got executed. We did not examine this binary any further. The last error injector does a probabilistic error injection. Error values are returned with a given probability per function call. We run 12 experiments with a fixed seed for reproducibility. One bug in the unpatched du results in a crash while our patch prevents this crash. Within wc none of the 4 found crashes are prevented by the patch. And even worse for grep our patch introduces an additional crash. We assume that the additional crash is a result of the preallocation. Our patch alters the ordering and the amount of memory reserved by the application. That is why a run with the patch behaves differently from a run without a patch.

Overall we found 79 crashes. Our generated patches prevent 64 of them – that are 81% of all crashes. This rate includes the additional crash of grep with the patch. We ran the same experiments with the early-exit patch. The numbers are the same except two additionally prevented crashes for du (with systematic and knockover error injection). So this patch prevents 66 of 79 crashes (84%).

Overhead Figure 5 shows the run time degradation of the applications with normal and early-exit patch relative to the run times without a patch. These runs are all done without error injection. The values are average values of 25 runs of each application with and without patches. The largest run time overhead is 9.14%. Surprisingly, the smallest is –4.88%, i.e., the generated patch accelerates the application. We conclude that our patches add little to no run time overhead.

Experiments indicate that the wrapper preloading is responsible for the speedup. Running touch with a wrapper that intercepts malloc without any additional actions takes in average 16 ms versus 21 ms for running touch without a wrapper.

4.2 Bugs Found

In this section we present a few of the bugs we have found with our tool. We explain the bugs and how the generated patch work to mask them on the concrete source code.

Missing Error Handling We found a common bug pattern in grep:

```c
/* grep −5.2.1 src/search.c:152 */
char *mb.properties = malloc (size);
mbstate_t cur_state;
wchar_t *wc;
int i;
memset(&cur_state, 0, sizeof(mbstate_t));
memset(mb_properties, 0, sizeof(char)*size);
```
Memory is reserved in line 152 and it is accessed without any error handling in line 157. The only thing the generated patch can do is preallocation. It tries to ensure that the unsafe call to malloc in line 152 always returns a pointer to a preallocated buffer. Of course, that might not be possible under all circumstances. It gets even more difficult because the argument of the unsafe call is dynamic. Our patch can prevent some crashes but we cannot guarantee that we can prevent all. The general solution is of course that the developer adds proper error handling code.

Error handling can be done in many ways. Two common strategies are:

- **Test Early**: This strategy should be sufficient for the bug described above.

- **Test before every use**: This could be the strategy the developers had intended for the next bug. But at least before one usage, the error handling is missing.

The next bug is in the hash table implementation used by du:

In function hash_initialize the memory for a hash table is reserved and initialized. While for the table itself a proper error handling is performed, the call to calloc in line 578 is not handled. We found no code that accesses table->bucket directly outside of the hash implementation. So it might work if before each use of table->bucket the hash implementation would check for an error value. But the next function in the source code file accesses the bucket without any check:

In function hash_clear, the bucket is initialized. While for the table itself a proper error handling is performed, the call to calloc in line 602 is not handled. We found no code that accesses table->bucket directly outside of the hash implementation. So it might work if before each use of table->bucket the hash implementation would check for an error value. But the next function in the source code file accesses the bucket without any check:
Bad Error Handling  The last bug we present tries to handle all possibly failing function calls:

```c
/* grep -5.2.1 src/dfa.c:3423 */
mp[i].in = (char **) malloc(sizeof *mp[i].in);
mp[i].left = malloc(2);
mp[i].right = malloc(2);
if (mp[i].left == NULL || mp[i].left == NULL ||
    mp[i].right == NULL || mp[i].is == NULL)
    ...
/* line 3623 */
done:
/* line 3634 */
for (i = 0; i < dfa->tindex; ++i) {
    freelist(mp[i].in);
    ifree((char *) mp[i].in);
    ifree(mp[i].left);
    ifree(mp[i].right);
    ifree(mp[i].is);
}
```

But the error handling itself contains a bug:

```c
93 /* grep -5.2.1 src/dfa.c:3240 */
static void freelist (char **cpp)
{
    int i;
    if (cpp == NULL) return;
    for (i = 0; cpp[i] != NULL; ++i) {
        free(cpp[i]);
        cpp[i] = NULL;
    }
}
```

The error handling starts in line 3427 directly after reserving some memory. If the reservation fails already reserved resources are freed and the current function returns. But what if only one of the three calls from line 3424 to line 3426 fails? The error handling code is executed and freelist tries to free an uninitialized list. Because mp[i].in is not initialized freelist walks outside of mp[i].in and the applications crashes (see for loop in line 3245 and line 3247).

A developer would possibly fix this bug by changing the call in line 3423 from malloc to calloc. So the list will be initialized and the for loop (line 3245) will not leave the list. But our static analysis finds a call group with one safe call (line 3423) and three unsafe ones (lines 3424–3426). The unsafe calls have static arguments. Thus, the error value mapping within a call group can be applied. When the call in line 3423 is issued, the memory of the next three calls is preallocated. When one of this four calls fail the patch returns an error value. This error value will be handled properly. For the other three calls it returns the preallocated memory. In this way the generated patch prevents the crash by mapping the error values from three unsafe calls to a safe one.

5 Conclusion

We have introduced a novel approach to detect and patch bad error handling. The basic idea is to use error injection to locate calls to library functions that do not perform proper error handling. We then use static analysis to determine if and what type of patch can be used to correct such a call. We show the effectiveness of our approach regarding several open source programs: we can reduce the number of potential crash failures without introducing any unacceptable performance penalties.

Future Work  For the future we plan to do a broader evaluation with more functions of interest and more applications. In particular, we want to explore possible side effects of our patch patterns. To make the approach more generally applicable, we need to do future research in patch patterns.

Further information on AutoPatch can be found on the AutoPatch homepage:
http://wwwse.inf.tu-dresden.de/AutoPatch/.

References