A Source-to-Source Transformation Tool for Error Fixing

Dr. Youry Khmelevsky*, Dr. Martin Rinard**, Dr. Stelios Sidiroglou-Douskos**

* Computer Science, UBC Okanagan, Kelowna, BC, Canada

** CSAIL, MIT, Cambridge, MA, USA

Abstract

We present a methodology and a prototype of a source-to-source transformation tool for error fixing in C/C++ program source code for missing condition checks after a method call. The missing condition checks in a C program could lead to a program crash. This tool can be extended for other programming languages in addition to C/C++.

The developed tool includes the ability to generate and apply a fix for a source code without human intervention. The tool can be run on different platforms, including MS Windows, Linux, MAC OS and other operating systems. We evaluate our technique by applying it to five widely used open source programs. Our results show that it is able to successfully detect and add the missing condition check or correct it after a method call in the program, and that our detection and error fixing technique is quite accurate in practice.

1 Introduction

A key component of any reliable software system is its exception handling, which allows the system to detect errors and react to them correspondingly [1]. A system without proper exception handling is likely to crash continuously, which renders it useless for practical purposes. For instance, more than 50% of all system failures in a telephone switching application are due to faults in exception handling algorithms [1]. Even the simplest exception handling strategy takes up to 11% of an application's implementation, and it is scattered over many different files and functions and is tangled with the application's main functionality [15, 16]. The C and Cobol programming languages do not explicitly support exception-handling, but are still used to develop and support software systems. The most common form of exception-handling method used by software programmers is the "return-code" technique that was popularized as part of C and UNIX, as it's shown in Figure 1 [30]:

```
1 if ((fd = open(filename, O_RDONLY)) == -1) {
2   fprintf (stderr, "Error %d opening file %s\n",
          errno, filename);
3   exit( );
4 }
```

Figure 1: The "return-code" technique example.

Although this technique is the most popular, it has many major drawbacks [30]:

1. Error Prone: Checking return values by if-statements is easy for programmers to ignore.
2. Poor Modular Decomposition: The main execution thread of the operation is mixed with if-statements and error-handling code.
3. Poor Testability: It is difficult to analytically verify that every possible error has a known handler, and it is hard to test every scenario in a systematic manner.
4. Inconsistency: The return value which denotes error is inconsistent. Some functions return NULL to indicate errors while others use -1.
5. Lack of Information: Any additional data crucial to handling exceptions must be passed outside the return code method.

In this paper we present a technique and a prototype of a source-to-source transformation

tool that detects missed checking of the return value of a method call and adds the missed checks or corrects them. Many programmers fail to incorporate error checking in specific classes of I/O operations where I/O errors could occur and where there is no mechanism in place to handle the error. They also rely on certain assumptions, such as “file output is always guaranteed", to ensure correct application operation. In many cases, software developers duplicate source code to replicate functionality, but this practice can produce additional bugs when two identical code segments are edited inconsistently [2]. The original code can have unfixed errors too.

Incorrect error handling is a longstanding problem in many application domains [3], but it is especially troubling when it affects I/O operations. File systems occupy a delicate middle layer in operating systems. For instance, a popular operating system such as Linux has many tools and applications that are written in C, which offer no exception handling mechanisms by which an error code could be raised or thrown. Errors must propagate through conventional mechanisms such as variable assignments and function return values. Correct error checking associated with an I/O routine must occur between the set (called a definition or simply def) of the potential error value and use of a result value or values along all possible paths of execution [4]. For instance, the C code example for the fopen() call of the stdio.h C library, which is shown in Figure 2 can lead to a program crash. The successful return of the fopen() call is a handle to a file. A zero, also referred to as NULL, should then be used to report that the file system was unable to open the file [4].

```
1 FILE * fp = fopen (sTmp, "w");
2 fprintf (fp,"%d", getpid ( ));
```

Figure 2: Code that may lead to a failure.

The code example in Figure 3 protects against a possible error condition (lines 2 - 5) [4].

```
1 FILE * fp = fopen (sTmp, "w");
2  if (fp != NULL) {
3    fprintf (fp, "%d", getpid( )) ;
5  }
```

Figure 3: Program logic guards against a possible unsuccessful result.

Many other source code examples with incorrect error handling or a missed condition check after a method call are available on the Internet, which could be found by using such code search engines as Google, Koders, Codase, etc. [4].

As we mentioned above, detecting and handling errors in system calls written in older programming languages such as C that do not explicitly support exceptions typically relies on an idiomatic approach for signaling and handling exceptions. Though C++ supports exceptions, C++ code may rely on legacy code and libraries that are written in C, and use the return code idiom [5]. Bruntink et al [1] showed that exception handling behavior is hard to test, as the root causes that invoke the exception handling mechanism are often difficult to generate. It is also very hard to prepare a system for all possible errors that might occur at runtime. The environment in which the system will run is often unpredictable, and errors may thus occur for which a system was not prepared. Moreover, they inform “any software system that is developed in a language without exception handling support will suffer the same problems". Mortensen et al described the following potential faults associated with the return codes [5]:

1. If the system does not check the return code of a function, the exception is ignored with potentially unpredictable behavior beyond that point.
2. The error code may need to be propagated up the call stack so that a series of calling functions can correctly check and signal exceptions through return values.
3. Contextual information may need to be passed from the location of the exception to the function that should handle it. Such information is often managed through global variables and log files and may not be consistently implemented throughout an application.

1.1 Usage Scenarios

We want to detect faults statically, because early detection and prevention of faults is less costly [28, 29], and because testing exception handling is inherently difficult [1].

The developed tool scans C/C++ source code and is capable of statically detecting violations to the return code idiom in the source code. We use

def-check-use analysis for the systems prototype, which was described by Bigrigg and Vos [4] for static analysis and source code correction. In the testing environment, before debugging C/C++ source code, the suspected code is deployed and evaluated with accordance to the reported problems. In a few cases we were unable to repeat the reported problems and we found that the problems were not confirmed by some other users/testers (for instance, the bug issue was reported once only and nobody could confirm it). In other cases, the version of the application was outdated, which can be incompatible with the modern OS version, or the current compiler version was incompatible with the old version of the application. The source code is then corrected by the tool, compiled, and re-deployed for testing purposes. We implement the following logical checks:

1. Checking of method calls against NULL or other possible values. In the current implementation we tested with the fopen() method only. The method calls, like fmpfile(), freopen(), fgets(), etc. can be checked by the tool in future releases.
2. Check for the pointer name in the corresponding check for the method call (if the check was found by the tool).
3. Check for the main method type in the source code to add appropriate return statements.

Some extra parameters will be described later in the following Sections.

1.2 Contributions

This paper makes the following contributions:

1. It presents a methodology for automatically detecting missed check(s) of the return value of a method call in a C/C++ source code.
2. A technique of automatically fixing errors in a source code. The developed prototype tool automatically fixes the system call error handling mechanism. The tool recognizes the type of the calling functions and then adds an appropriate return statement.
3. Experimental results: It presents experimental results that characterize how well the technique works on five applications drawn from the open-source software community. The results show that our technique can detect and fix missed checks of the return value of the method calls, and/or correct appropriate pointer names in the source code as well.

2 Related Works

In this section we discuss related works in source code checking and source code error fixing. As it is discussed in [24], the "current static code analysis tools are able to detect errors in programs, but most cannot actually fix the errors. Manual debugging is necessary to fix these issues, but developers make mistakes and often work under tight deadlines".

2.1 Source Code Checking

Marri et al [6] proposed a life-cycle model that can be used to develop approaches based on code searching and mining in the two phases of the life-cycle model. They also suggested post-processing techniques for mining patterns from gathered code examples, which can be used to detect defects in a program under analysis. Furthermore, they demonstrate the application of their life-cycle model with a preliminary evaluation. The developed PARSEWeb, which includes 16 heuristics. These heuristics are contrary to the type checking done by a compiler. PARSEWeb accepts queries of the form "Source object type → Destination object type" and finds method-invocation sequences that produce the destination object type from the source object type. Our prototype is developed to fix defects automatically in a program under analysis.

Bigrigg and Vos [4] presented a methodology that detects robustness failures in source code where I/O errors could occur with no mechanism in place to handle the error. The presented methodology, based upon a static analysis of the program, is to track the propagation of error reporting in order to determine the assumptions used when the software was created. A dataflow analysis was described for detecting bugs in the propagation of errors in user applications. It augments traditional def-use chains with intermediate check operations. A working implementation that is interprocedural and context-sensitive has been applied to thousands of lines of kernel code and detection of the overwritten, out-of scope, and unsaved unchecked

errors on Linux file system implementations [7]: CIFS, ext3, IBM JFS, ReiserFS, ext4, and shared virtual file system (VFS). Flow- and context-sensitive approaches produced results while providing diagnostic information, including possible execution paths that demonstrate each bug found. For the implementation, the CIL C front end was used [8]. The WALi WPDS library [9] was used to perform the interprocedural dataflow analysis on the WPDS. Within the WALi-based analysis code, weights utilizing using binary decision diagrams (BDDs) were used [10].

Mortensen and Ghosh [5] used an aspect-oriented approach for throwing exceptions in place of the “return code idiom" and discussed the use of aspects to modularize scattered code for detecting and handling errors in system calls, such as fopen. Several potential faults associated with the return code idiom, which are often managed through global variables and log files and may not be consistently implemented throughout an application, are described [5]:

1. If the system does not check the return code of that function, the exception is ignored with potentially unpredictable behaviour beyond that point. This is a main case of our research and our findings in open source application and tools, published on the Internet.
2. The error code may need to be propagated up the call stack so that a series of calling functions must correctly check and signal exceptions through return values. This is a possible problem, but uncommon in our findings.
3. Contextual information may need to be passed from the location of the exception to the function that should handle it.

They manually refactored the PowerAnalyzer to use aspects for throwing and catching exceptions. This aspect provides a modular way of adding exceptions using a pointcut that is easy to specify and maintain since it is based only on the name of the function (fopen) that triggers the error. In our research our prototype tool performs static analysis of program source code and adds missing condition checks after a method call automatically.

ITS4, a token-based scanning tool for statically scanning C and C++ source code for security vulnerabilities [11], offers real-time feedback to developers during coding and scanning C++ code. ITS4 breaks a non-preprocessed file into a series of lexical tokens, and then matches patterns in that stream of tokens. Matching code and then matching patterns in that stream of tokens. Matching code is added by hand, so non-regular patterns can be recognized [11]. They identified several problems related to Advanced Static Analysis for C/C++ and informed that C's liberal nature makes the language poorly suited for static analysis. They also addressed race conditions in file accesses, so-called “Time-Of-Check, Time-Of-Use” (TOCTOU) problems. We use these in our tool as well. The TOCTOU functions are classified, based on their handler, into functions that can be checks and functions that can be used. Every time they see a function, they look at the identifier that holds the file name. They store a mapping of variables to the list of TOCTOU functions that use that variable, but they do not address the aliasing problem.

Bruntink et al [1] analyzes the exception handling mechanism of an industrial embedded software system (developed by ASML, a Dutch company) that uses the return code idiom for dealing with exceptions. In the Related Work section they discussed Fault (Bug) Finding techniques and Metal [12], PREfix [13], and ESC [14] tools, and the related model checking CMC tool [15]. Additionally, they discussed program verification, which is focused on proving specified properties of system tools (MOPS [16], SLAM [17] and ESP [18]), and Idiom Checkers, that can find basic coding errors [19], [20]. But, as they inform, these tools are incapable of verifying domain-specific coding idioms, such as the return code idiom [1]. On the other hand, more advanced tools [21], [22] are restricted to detecting higher-level design flaws but are not applicable at the implementation level [1]. The focus of the paper was to analyze which faults can be introduced and to show how they can be detected and prevented. Based on the fault model, they developed SMELL, the State Machine for Error Linking and Logging, which is capable of statically detecting violations to the return code idiom in the source code, and is implemented as a CodeSurfer plugin [36]. Their approach has limitations, both formally unsound and incomplete [1]: both false negatives (missed faults) or false positives (false alarms) are possible, but as they inform, the unsoundness property and incompleteness properties do not necessarily harm the usefulness of their tool, given that the tool still allows to detect a large number of faults

that may cause much machine down-time, and that the number of false positives remains manageable.

The DynaMine tool [23] analyzes source code check-ins (revision histories) to find highly correlated method calls as well as common bug fixes in order to automatically discover application-specific coding patterns. The combination of revision history mining and dynamic analysis techniques leveraged in DynaMine is effective for both discovering new application-specific patterns and for finding errors when applied to very large applications. They have analyzed Eclipse and jEdit Java applications. Their technique for mining patterns from software repositories can be used independently with a variety of bug finding tools by looking for pattern violations at runtime, as opposed to using a static analysis technique. They inform that certain categories of patterns can be gleaned from antipattern literature, but many antipatterns tend to deal with high-level architectural concerns rather than with low-level coding issues. The idea to use patterns and antipatterns is interesting, but for many typical errors in the source code, a more simple way can be used to fix the common problems. One can simply add a missed check to the return code of a method call. Source code error fixing automatic patch generation was described by Michael Lam in [24], but he says that current techniques are limited to well-defined scopes and problem domains, and they can only suggest good solutions to very specific types of problems, usually involving common programming mistakes. Future work may involve extending current techniques, integrating tools from related fields, and justifying the usefulness of patch generation with empirical studies. She says that it is ultimately impossible to always generate perfect patches.

McAdam in [25-26] describes a system for fixing type errors in functional programs. This system finds possible replacements for type-unsafe expressions by rewriting them according to currying and associativity axioms, performing associative-commutative unification on the resulting forms, and finally performing a partial evaluation to achieve a human-readable and useful output. McAdam implemented this system in a tool for Mlj.

Weimer [27] describes a system for generating patches from static analysis error reports. This tool is able to automatically generate missing code or remove extraneous code to produce a program that satisfies a given policy. Weimer also includes a study of the effectiveness of this method across several projects by reporting bug fix rates as correlated with automatic patch suggestion.

Several projects have used the term “automatic patch generation” to describe the process of repairing binaries after a malicious attack such as a buffer overflow [24], [28-30].

Error correcting compilers [31], [32] actually serve a slightly different purpose. The main goal of an error-correction routine in a compiler is to continue the compilation process even after finding a syntactic error, and there are few attempts to find or fix semantic or logical errors [24].

AutoPaG [33] aims at reducing the time needed for software patch generation. It focused on the out-of-bound vulnerability, which includes buffer overflows and general boundary condition errors. The AutoPaG is able to catch the out-of-bound violation, and then, based on data flow analysis, automatically analyzes the program source code and identifies the root cause – vulnerable source-level program statements. AutoPaG generates a source code patch to temporarily fix it without human intervention.

Hovemeyer and Pugh [34] demonstrate automatic detector implementation for a variety of bug patterns found in Java programs. They have found that the effort required to implement a bug pattern detector by using relatively simple static analysis techniques tends to be low, and that even extremely simple detectors find bugs in real applications. They have found that even well tested code written by experts contains a surprising number of obvious bugs and even Java (and similar languages) has many language features and APIs, which are prone to misuse. A simple automatic technique can be effective at countering the impact of both ordinary mistakes and misunderstood language features inspection. Authors have implemented a number of automatic bug pattern detectors in a tool called FindBugs [37]. They discuss dereferencing a null pointer, which almost always indicates an error. The detector for this pattern catches many obvious null dereferences errors, but they don’t discuss any automatic bug fixing implementation for the null pointer exceptions to compare with our tool. The tool was developed for Java, not for the C language, as in our research.

Microsoft’s PREfast is a similar utility for static code analysis for MS Windows family of

OSs. “It can find defects in C/C++ code such as buffer overruns, null pointer dereferencing, forgetting to check function return value and so on” [51]. PREfast is a tool for an automatic code review, but not for automatic bugs fixing by source-to-source code transformation and doesn’t work on other operating systems to compare with our OS agnostic prototype.

The approach described in [52] uses legality assertions, source code assertions inserted before each subscript and pointer dereference that explicitly check that the referencing expression actually specifies a location within the array or object pointed at run time. They have developed a transformation system to automatically insert legality assertions in the source program where array elements are accessed or pointers are dereferenced. In our transformation tool we used conditions checks instead of assertions.

3 A Model for the Tool

To distinguish different components of the return code idiom in our tool, we developed a model for missed exception handling within the return code idiom [1], the exception handling mechanism (EHM) [35] and the *def-check-use* analysis [4].

The Open Group Base Specifications Issue 6 IEEE Std 1003.1, 2004 Edition explains that “Upon successful completion, fopen() shall return a pointer to the object controlling the stream. Otherwise, a null pointer shall be returned, and errno shall be set to indicate the error” [38].

Our model defines several different error scenarios, which can be automatically fixed by the developed tool. We investigated many C/C++ programs and tools when a method was called without exception handling and when program crashed during execution in our testing environment. We found references on the Internet for such problems and some of them are described in the Evaluation section. We found two main problems:

1. A missed error detection (or check) in the source code. In our case, a missed checking of the value of the method call against NULL or another value. See example in Figure 2 and the corrected source code example in Figure 3.
2. Incorrect error detection (or check) in the source code.

The return code idiom [1] relies on the fact that when an error is received, the corresponding error value should be logged and should propagate to the return statement. In our model, the return code idiom relies on the fact that when an error is received from the call function, the corresponding error value should be evaluated (checked) and error notification should propagate to the return statement (returned), depending on the type of the main or calling function (void, int, bool or char methods).

A call function can be regarded as a black box, i.e., only its input–output behaviour is considered. Any error values received from the call functions (*receive* predicate in Table 1) are regarded as input. Outputs are comprised of the error value that is returned by a main or calling function (*return*) [1].

The fault model makes two simplifications. First, it assumes that a function receives at most one error during its execution. Second, only one error value can be received, it makes little sense to link more than one error value to it [1].

Our fault model consists of three categories. Each includes a failure scenario. We define several potential faults associated with the return code idiom [2]:

1. Category 1. The received error value x is not checked. The exception is ignored with potentially unpredictable behaviour beyond that point (see example in Figure 2).
2. Category 2. The received error value x is not checked, but the value y is checked instead. This is a possible human error, when the part of source code can be copied from examples from the Internet or internal code repositories. The pointer names are different (see lines 1, 2, and 6 in example in Figure 4). The exception is ignored with potentially unpredictable behaviour beyond that point.

   Possible problems include erroneously checked error value y.
3. Category 3. The received error value x is not checked, but the value y is checked instead, the exception is ignored and the value y is used with potentially unpredictable behaviour beyond that point (see example in Figure 5).

The predicates capturing the faults in each category are displayed in Table 1. Just as an example the last column in the table (Correct) shows correct error handling for a function, where *z* is returning error message to the calling function (void for the void type of the calling function, -1 for the int, -1 for the float, -1 for double, False for the bool if the bool datatype and False were declared in the C application explicitly, and 1 for the char datatype).

Table 1. Predicates for the three fault categories and the last one show correct error handling.

Category 1	**Category 2**	**Category 3**	**Correct**
receive (x) ^ use (x)	receive (x) ^ check (y) ^ use (x) ^ x ≠ y	receive (x) ^ check (y) ^ use (y) ^ x ≠ y	receive (x) ^ check (x) ^ return (z) ^ z = { Void, False, -1, 1 for char}

```
1 FILE * fp1 = fopen (sTmp , "w");
2 if (fp2 == NULL) {
3    printf ("Error while opening file! \n");
4    return -1;
5 }
6 fprintf (fp1, "%d", getpid ( ));
7 fclose (fp1);
```

Figure 4: Category 2 possible fault associated with the return code idiom.

```
1 FILE * fp1 = fopen(sTmp, "w");
2 if (fp2 == NULL) {
3    printf ("Error while opening file! \n");
4    return -1;
5 }
6 fprintf(fp2,"%d", getpid ( ));
7 fclose(fp2);
```

Figure 5: Category 3 possible fault associated with the return code idiom.

4 Example

Figure 6 presents a snippet of the Rsync utility source code for the Rsync version 2.6.3, protocol version 29 that may lead to a failure [39]. At line 147, the fopen() method was used without checking the value of the method call against NULL. As we discussed in the Section 1, such source code can lead to a program crash, or as it was reported [40]: "... got a core dump when starting a daemon which tried to write to a log file that it had no permission to write to."

```
142 void log open(void)
143 {
144   if (logfname && !logfile) {
145    extern int orig_umask;
146         int old_umask = umask (022 | orig_umask) ;
147    logfile = fopen(logfname , "a ") ;
148    umask (old_umask) ;
149   }
150 }
```

Figure 6: The Original Snippet of Rsync Utility V.2.6.3, Protocol Version 29.

```
1 $ sudo rm /var/log/rsyncd.log
2 $ sudo mkdir /var/log/rsyncd.log
3 $ sudo /usr/local/bin/rsync --daemon -v --no-detach
4 $ sudo tail -f /var/log/messages
5 . . .
6 Dec 3 13:45:58 abrt [6491]: saved core dump of pid
  6490 (/usr/local/bin/rsync) to /var/spool/abrt/ccpp-
  1291401958-6490.new/coredump (425984 bytes)
7 Dec 3 13:45:58 abrtd: Directory 'ccpp-1291401958-
  6490'  creation detected
8 Dec 3 13:45:58 abrtd: Executable
  '/usr/local/bin/rsync' doesn't belong to any package
9 Dec 3 13:45:58 abrtd: Corrupted or bad crash
  /var/spool/abrt/ccpp -1291401958-6490 (res:4),
  deleting
```

Figure 7: The Core Dump Registration with Rsync Utility Version 2.6.3, Protocol Version 29.

We deployed the Rsync utility on a Linux host and tested with different Rsync configuration options. We indeed registered a core dump in the system /var/log/messages, as it's shown in lines 6-7 on Figure 7.

The developed tool corrected the Rsync utility source code automatically, by adding lines 147-156, as it is shown in Figure 8. As a result of the automatic source code debugging, we had no core dump anymore and the Rsync utility was successfully running in the same testing environment, but the logfile was written into the local logfile instead of default location.

Note: In this example the additional option for the tool was used to implement the spoofing file in the local folder.

```
142 void log open (void) {
143   if (logfname && ! logfile) {
144     extern_int orig_umask ;
145     int old umask = umask (022 |orig_umask);
146     logfile = fopen(logfname ,"a ") ;
147          /* filehandler check for NULL was added
           automatically */
148        if(logfile == NULL) {
149          printf("Error while opening file!\n");
150          // the file is replaced by ./TempFile
151            logfile = fopen("TempFile", "a");
152            if(logfile == NULL) {
153              printf("Error while opening the TempFile
                       file too , sorry!\n" ) ;
154              return;
155            }
155          return;
156       }
156   /*end of the filehandler check for NULL code! */
157    umask ( old_umask ) ;
158   }
159 }
```

Figure 8: The Fixed Snippet of Rsync V.2.6.3.

5 Implementation

Our automatic error fixing technique contains three main components:

1. The first logical component of the tool searches for all method calls in the program source code by a pattern match. In the prototype the fopen() method call was used for the testing purposes only, but the prototype can be extended for other method calls as well.
2. The second logical component detects missed parts of the source code, which should check the value of the method calls against NULL, but can be extended for other values as well.
3. The third logical component adds the following missed parts of the code or corrects pointer names in the following part of the source code:
    a. Checks the value of the method calls against NULL (see example on the Figure 8 lines 147-156).
    b. Corrects any mistyped FILE type pointer names.
    c. Evaluates the type of the main function, which contains the method call to use appropriate return statement. See lines 142 and 154 in the example on the Figure 8.
    d. If the additional “--spoofing" option was used, the tool uses a “spoofing" filename to open a file in the current location (see “logfile" pointer name as an example of where this problem could occur and be fixed on the Figure 8 lines 146 and 148). This option was found useful in one testing example, which is shown in Figure 5 to avoid a core dump or an application crash.
    e. Finally, the tool updates the program source code.

6 Evaluation

We evaluate our technique by applying it to several sizable, widely-used programs selected from the open-source software community with publicly reported 8 bugs of the fault category 1 (see Section 3). These programs include:

1. Rsync, which is an open source utility that provides fast incremental file transfer. Rsync is freely available under the GNU General Public License and is currently being maintained by Wayne Davison [41].
2. Gtk-theme-switch, which is a small and fast command line utility to switch GTK themes on the fly. It also has an optional GUI to preview the requested theme and change the font used with it, an optional GUI dock, and it can install themes downloaded from gtk.themes.org, preview them, or switch to them immediately [42].
3. Gtk-chtheme, a program that lets you to change the gtk+2.0 theme. The aim is to make theme preview and selection as slick as possible. Themes installed on the system are presented for selection and previewed on the fly [43].

4. Gammu, the name of the project as well as name of a command line utility, can be used to control a cell phone. It is written in C and built on top of libGammu. The Gammu command line utility provides access to wide range of phone features, however support level differs from phone to phone and you might want to check Gammu Phone Database for user experiences with various phones [44].
5. AbiWord, which is a free word processing program similar to Microsoft Word. It is suitable for a wide variety of word processing tasks. AbiWord allows you to collaborate with multiple people on one document at the same time. It is tightly integrated with the AbiCollab.net web service, which lets you store documents online, allows easy document sharing with your friends, and performs format conversions on the fly.

We used available regression tests for the AbiWord, Gammu and Rsync to check our error fixing by the tool. In the cases when we had no regression tests, we tested applications with typical application operations manually. All of these programs may execute, in principle, for an unbounded amount of time. Rsync, in particular, is typically deployed as a part of a standard computing environment.

All of the programs mentioned above, which were deployed from original source code, crashed in the reported environments or with specific configuration files (see appropriate references and additional information in the related subsections below). All applications continued to run in the same environment and with the same configuration parameters after reported bugs were fixed automatically by the tool. We applied our tool for the specific source files only, which appeared in the bug report. Some files had many hundred lines of code, but some of them had relatively small size. The tool was able to fix several places in the same source files, where errors were identified. On the other hand, we tested our tool against already fixed code, and against code, which has none of the problems mentioned above as well. The tool didn't report any problems against corrected or well-written code.

The identification and the fixing took from several milliseconds to several seconds for a largest files on a Quad Core Intel based CentOS (Linux) with 3 GHz CPU for all investigated applications. We didn't investigate performance, because it took considerably small amount of time for all tested applications.

Our evaluation focuses on the two following issues:

1. The ability of our technique to find missed or incorrect parts of source code, which should check the value of a method call (by a specific pattern).
2. The automatic insertion of missed parts of source code or on correction of the mistyped pointer in the source code (check the value of the method calls with appropriate type of the return statement in our case).

We perform the following experiments for each program:

1. Searching and Automatic Correction Runs: We used our tool to evaluate a piece of source code and automatically add missed parts of the source code, which should check the value of a method call.
2. Validation Runs: We've built and deployed the original and updated programs to confirm the reported bugs and check the results of the bug fixing by the tool.

As an example of our evaluation (see more in the Section 4 above), the Figure 6 presents a snippet of the Rsync utility that may lead to a failure [39]. At line 147, the fopen() method was used without checking the value of the method call against NULL. We deployed the Rsync utility and tested with different Rsync configuration options. As we mentioned above, we indeed registered a core dump in the system /var/log/messages, as it's shown in lines 6-7 on Figure 7. Our tool corrected the Rsync utility source code automatically, by adding lines 147-156, as it is shown in Figure 8. In the same way we evaluated other four applications additionally to Rsync (see below).

6.1 Rsync

The evaluation was performed with version 2.6.3, protocol version 29. The bug was reported in the "rsync user list" of the samba.org mailing list on 28 January 2005 [46]. If the Rsync daemon was started with the incorrect log file name in the con-

figuration file (in our testing case a directory name was used instead of a log filename), the Rsync crash and cause a core dump, as it's shown in lines 6-7 on the Figure 7. The snippet of the original source code is shown on the Figure 6 and automatically fixed snippet is shown on the Figure 8, which allows running Rsync tool in the same environment with incorrect log file name without a core dump and Rsync crash.

6.2 Gtk-theme-switch

The evaluation was performed with the Gtk-theme-switch version 2.0.5. The bug was reported in the "debian-qa-packages" of the Debian mailing list on 10 September 2008 [47]. We performed the evaluation with the incorrect location of the configuration file. The switch crashed during our evaluation when we tried to save configuration parameters in a wrong location. Snippets of the original source code with missed checks of the return value of the fopen() method calls (4 method calls without appropriate checks) are shown on the Figure 9.

```
. . .
FILE *gtkrc = fopen(path ,"w") ;
fprintf (gtkrc , "# -- THEME AUTO-WRITTEN DO
NOT EDIT\n include \"%s \"\n\n", include_file);
. . .
FILE *gtkrc = fopen(path_to_gtkrc, "r ");
. . .
FILE *gtkrc = fopen(path, "w");
. . .
FILE *gtkrc_backup = fopen(g_strdup_printf(
"%s/.gtkrc-2.0.bak ", homedir),"w");
```

Figure 9: The Original Snippets of GTK Theme Switch V. 2.0.5.

After fixing errors automatically with the tool, the switch continue to run even with an incorrect configuration file location and a warning message in the terminal window during attempt to save configuration parameters in our tests. The fixed snippets of the source code are shown on Figure 10.

```
. . .
FILE * gtkrc = fopen(path, "w");
if(gtkrc == NULL){
  printf("Error while opening file!\n");
  return;
}
fprintf(gtkrc, "# --THEME AUTO-WRITTEN
   DO NOT EDIT\n include \"%s \"\n\n", include_file);
. . .
FILE * gtkrc = fopen(path_to_gtkrc, "r ");
if( gtkrc == NULL){
  printf("Error while opening file! \n");
  return -1;
}
. . .
FILE * gtkrc = fopen(path, "w");
if(gtkrc == NULL){
  printf("Error while opening file! \n");
  return;
}
. . .
FILE gtkrc backup = fopen (g_strdup_printf(
"%s/.gtkrc-2.0.bak" ,homedir ),"w");
if( gtkrc backup == NULL ){
  printf(" Error while opening file! \n");
  return;
}
```

Figure 10: The Fixed Snippets of GTK Theme Switch V. 2.0.5 by the Tool.

6.3 Gtk-chtheme

The evaluation was performed with the Gtk-chtheme version 0.3.1 [48]. The bug was reported in the Debian Bug report logs - #500076 "[gtk-chtheme] An unchecked fopen leads to SIGSEGV" on Wed, 24 Sep 2008 [49]. As it is informed in the report: "Like all gtk theme switching utility based on gtk-theme-switch, gtk-chtheme includes some unchecked getenv() calls. Add to this an unchecked fopen or fdopen, then it generates a SIGSEGV". The chtheme crashed during our evaluation when we tried to save configuration parameters in an incorrect location. The original snippet of the source code with the unchecked fopen is shown on the Figure 11.

```
…
void apply_new_look( gboolean is_preview )
{
if(! themename) return;
cleanup_temporary( );
FILE *gtkrcfh = is_preview ? fdopen (
g_file_open_tmp
( "gtkrc . preview-XXXXXXXX" , &tmp_rc , NULL) ,
"w+" ):
fopen( gtkrc , "w" ) ;
. . .
```

Figure 11: The Original Snippet of Gtk-chtheme V. 0.3.1-3.

After bugs were fixed by the tool (see the fixed snippet on the Figure 12), the Gtk-chtheme continued to run even with the incorrect configuration file location and a warning message appeared in the terminal window during an attempt to save the configuration parameters.

```
. . .
void apply_new_look (gboolean is_preview)
{
 if (! themename) return ;
 cleanup temporary ( ) ;
 FILE *gtkrcfh = is_preview ? fdopen (
g_file_open_tmp("gtkrc.preview-XXXXXXXX"          ,
&tmp_rc , NULL) , "w+") : fopen ( gtkrc , "w") ;
 if ( gtkrcfh == NULL)
 {
 printf ( "Error while opening file! \n" );
 return;
 }
. . .
```

Figure 12: The Fixed Snippet Gtk-chtheme V. 0.3.1-3 by the Tool.

6.4 Gammu

The evaluation was performed with Gammu version 1.17.90 to store the VCARD21 into the non-existent folder output le1.vcf. The Debian Bug report #463013: states “gammu exits with segmentation fault if trying to write a vcf file with savefile command" [50]. We confirmed this during our evaluation test with the original source code. The snippet of the original source code of the Gammu tool with the mentioned above bug is presented on the Figure 13.

```
…
file = fopen (argv [ 3 ] , "wb");
if ( j != fwrite (Buffer, 1 , j , file)){
 printf_err (_( "Error while writing file ! \ n ") ) ;
}
fclose (file) ;
. . .
```

Figure 13: The Original Snippet of Gammu V.1.17.90**.**

After bug fixing with the tool, Gammu continues to run even with the non-existing folder and a warning message in the terminal window during attempt to save output file 1.vcf. The snippet, which was fixed by the tool, is shown on the Figure 14.

```
. . .
file = fopen ( argv [ 3 ] , "wb" ) ;
if( file == NULL )
{
  printf ( "Error while opening file ! \n" ) ;
  return;
}
if ( j != fwrite ( Buffer , 1 , j , file )) {
  printf_err ( ( "Error while writing file ! \n" ) ) ;
}
fclose (file);
. . .
```

Figure 14: The Fixed Snippet of Gammu V. 1.17.90 by the Tool.

6.5 AbiWord

The evaluation was performed with AdiWord version 2.4.3 by using command line AbiCommand plugin, which failed towritepid with NULL value. The AdiWord crashed during our evaluation when we tried to execute writepid with NULL value in AbiWord command shell. The snippet of the original code of the AbiCommand tool is presented on the Figure 15 and the fixed snippet by the tools is shown on the Figure 16.

After the bug fixing, AbiWord doesn't crash and continues to run even with the warning message in the terminal window during attempt to execute the writepid with the NULL value.

```
. . .
{
  if ( pToks->getItemCount ( ) < 2) {
    return -1;
  }
UT String pidFile ;
pidFile = * constcast<UT_String *>(
static_cast<const UT_String *>(pToks->
getNthItem ( 1 ) ) ) ;
FILE * pidF = fopen ( pidFile.c_str( ) , "w" ) ;
fprintf ( pidF , "%d" , getpid ( ) ) ;
fflush ( pidF ) ;
fclose ( pidF ) ;
return 0 ;
}
. . .
```

Figure 15: The Original Snippet of AbiWord V.2.4.3 AbiCommand Plugin.

```
. . .
{
 if ( pToks->getItemCount ( ) < 2) {
   return -1;
 }
 UT_String pidFile ;
 pidFile = * const_cast<UT_String *>(
 static_cast <const UT_String *>(
 pToks->getNthItem ( 1 ) ) ) ;
 FILE * pidF = fopen ( pidFile.c_str ( ) , "w" ) ;
 if ( pidF == NULL ){
   printf ( "Error while opening file ! \n" ) ;
   return -1;
 }
 fprintf ( pidF , "%d" , getpid ( ) ) ;
 fflush ( pidF ) ;
 fclose ( pidF ) ;
 return 0;
}
```

Figure 16: The Fixed Snippet of AbiWord V. 2.4.3 AbiCommand Plaugin by the Tool.

6.6 Summary

In each discussed program from 1 to 4 condition checks were added by the tool (4 condition checks in Gtk-theme-switch version 2.0.5 and 1 condition check in other 4 programs). All of these condition checks belong to fault category 1. The analysis and repair took just a few milliseconds, because we analyzed and fixed only publicly reported bugs in the specific program files.

7 Conclusion

Incorrect error handling or the absence of error handling is a longstanding problem in many application domains, but it is especially troubling when it affects I/O operations.

Our automatic error fixing technique searches for missed checks on the value of the method calls or incorrect pointer names in source code and adds or corrects them automatically. Our results show that this technique can eliminate important problems in programs, which lead to crashes and/or core dumps.

Acknowledgments

Our thanks to Okanagan College Extended Study Leave committee for the financial support and active supervision of this research project in the Computer Science and Artificial Intelligence Laboratory (CSAIL) at Massachusetts Institute of Technology (MIT).

The authors would like to thank the reviewers for their valuable comments, which have helped to improve this paper.

About the Authors

Dr. Youry Khmelevsky is a Professor in Computer Science at Okanagan College, Kelowna, BC from 2005 and Adjunct Professor in Computer Science at UBC Okanagan, Kelowna, BC, Canada from 2011. He can be reached at the address Computer Science, Science Health and Technology School, 1000 KLO Rd., Kelowna, BC, V1Y 4X8, Canada. His Internet addresses are ykhmelevsky at okanagan dot bc dot ca and youry dot khmelevsky at ubc dot ca.

Dr. Martin C. Rinard is a Professor in the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology, and a member of the Computer Science and Artificial Intelligence Laboratory. He can be reached at the address MIT Computer Science and Artificial Intelligence Laboratory, the Stata Center, Building 32-G828, 32 Vassar Street Cambridge, MA 02139. His Internet address is rinard at lcs dot mit dot edu.

Dr. Stelios Sidiroglou-Douskos is a research scientist in the Computer Science and Artificial Intelligence Laboratory at MIT in Cambridge, MA. He is also a member of the Center for Reliable Software CRS. His Internet address is stelios **at** csail dot mit dot edu.

References

[1] M. Bruntink, A. van Deursen, and T. Tourwé, “Discovering faults in idiom-based exception handling,” 2006, p. 242.

[2] M. Gabel, J. Yang, Y. Yu, M. Goldszmidt, and Z. Su, “Scalable and systematic detection of buggy inconsistencies in source code,” 2010, p. 175.

[3] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau, “Error propagation analysis for file systems,” 2009, p. 270.

[4] M. W. Bigrigg and J. J. Vos, “The set-check-use methodology for detecting error propagation failures in I/O routines,” WDB’02, 2002.

[5] M. Mortensen and S. Ghosh, “Refactoring idiomatic exception handling in C++: Throwing and catching exceptions with aspects,” URL http://aosd.net/2007/program/industry/ I, vol. 2, 2007.

[6] M. R. Marri, S. Thummalapenta, and T. Xie, “Improving software quality via code searching and mining,” in Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation, 2009, pp. 33–36.

[7] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau, “Error propagation analysis for file systems,” SIGPLAN Not., vol. 44, no. 6, pp. 270–280, Jun. 2009.

[8] G. Necula, S. McPeak, S. Rahul, and W. Weimer, “CIL: Intermediate language and tools for analysis and transformation of C programs,” in Compiler Construction, 2002, pp. 209–265.

[9] N. Kidd, T. Reps, and A. Lal, WALi: A C++ library for weighted pushdown systems. 2009.

[10] R. E. Bryant, “Binary decision diagrams and beyond: Enabling technologies for formal verification,” in Computer-Aided Design, 1995. ICCAD-95. Digest of Technical Papers., 1995 IEEE/ACM International Conference on, 1995, pp. 236–243.

[11] J. Viega, J. T. Bloch, T. Kohno, and G. McGraw, “Token-based scanning of source code for security problems,” ACM Transactions on Information and System Security, vol. 5, no. 3, pp. 238–261, Aug. 2002.

[12] D. Engler, B. Chelf, A. Chou, and S. Hallem, “Checking system rules using system-specific, programmer-written compiler extensions,” in Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4, 2000, pp. 1–1.

[13] W. R. Bush, J. D. Pincus, and D. J. Sielaff, “A static analyzer for finding dynamic programming errors,” Software-Practice and Experience, vol. 30, no. 7, pp. 775–802, 2000.

[14] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, “Extended static checking for Java,” in Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, New York, NY, USA, 2002, pp. 234–245.

[15] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill, “CMC: A pragmatic approach to model checking real code,” ACM SIGOPS Operating Systems Review, vol. 36, no. SI, pp. 75–88, 2002.

[16] H. Chen and D. Wagner, “MOPS: an infrastructure for examining security properties of software,” in Proceedings of the 9th ACM conference on Computer and communications security, 2002, pp. 235–244.

[17] T. Ball and S. K. Rajamani, “The S LAM project: debugging system software via static analysis,” ACM SIGPLAN Notices, vol. 37, no. 1, pp. 1–3, 2002.

[18] M. Das, S. Lerner, and M. Seigle, “ESP: Path-sensitive program verification in polynomial time,” ACM SIGPLAN Notices, vol. 37, no. 5, pp. 57–68, 2002.

[19] S. C. Johnson, “Lint, a C program checker,” Computer science technical report, vol. 65, 1978.

[20] S. Paul and A. Prakash, “A framework for source code search using program patterns,” Software Engineering, IEEE Transactions on, vol. 20, no. 6, pp. 463–475, 1994.

[21] E. Van Emden and L. Moonen, “Java quality assurance by detecting code smells,” in Reverse Engineering, 2002. Proceedings. Ninth Working Conference on, 2002, pp. 97–106.

[22] T. Tourwé and T. Mens, “Identifying refactoring opportunities using logic meta programming,” in Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on, 2003, pp. 91–100.

[23] B. Livshits and T. Zimmermann, “DynaMine: finding common error patterns by mining software revision histories,” SIGSOFT Softw. Eng. Notes, vol. 30, no. 5, pp. 296–305, Sep. 2005.

[24] M. Lam, “Automatic Patch Generation,” 2007.

[25] B. J. McAdam, “How to repair type errors automatically,” Trends in functional programming, vol. 3, p. 87, 2002.

[26] B. J. McAdam, “Repairing type errors in functional programs,” 2002.

[27] W. Weimer, “Patches as better bug reports,” in Proceedings of the 5th international conference on Generative programming and component engineering, 2006, pp. 181–190.

[28] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, and others, “Automatically patching errors in deployed software,” 2009.

[29] A. Smirnov and T. Chiueh, “Automatic patch generation for buffer overflow attacks,” in Information Assurance and Security, 2007. IAS 2007. Third International Symposium on, 2007, pp. 165–170.

[30] S. Sidiroglou and A. D. Keromytis, “Countering network worms through automatic patch generation,” Security & Privacy, IEEE, vol. 3, no. 6, pp. 41–49, 2005.

[31] A. V. Aho and T. G. Peterson, “A minimum distance error-correcting parser for context-free languages,” SIAM Journal on Computing, vol. 1, p. 305, 1972.

[32] P. Degano and C. Priami, “Comparison of syntactic error handling in LR parsers,” Software: Practice and Experience, vol. 25, no. 6, pp. 657–679, 1995.

[33] Z. Lin, X. Jiang, D. Xu, B. Mao, and L. Xie, “AutoPaG: towards automated software patch generation with source code root cause identification and repair,” in Proceedings of the 2nd ACM symposium on Information, computer and communications security, 2007, pp. 329–340.

[34] D. Hovemeyer and W. Pugh, “Finding bugs is easy,” SIGPLAN Not., vol. 39, no. 12, pp. 92–106, Dec. 2004.

[35] M. Lippert and C. V. Lopes, “A study on exception detection and handling using aspect-oriented programming,” in Software Engineering, 2000. Proceedings of the 2000 International Conference on, 2000, pp. 418–427.

[36] http://www.grammatech.com

[37] http://findbugs.sourceforge.net

[38] http://pubs.opengroup.org/onlinepubs/009695399/functions/fopen.html

[39] http://samba.anu.edu.au/ftp/rsync/src/rsync-2.6.3.tar.gz

[40] http://lists.samba.org/archive/rsync/2005-January/011472.html

[41] http://samba.anu.edu.au/rsync/

[42] http://freshmeat.net/projects/gtkthemeswitch/

[43] http://plasmasturm.org/code/gtk-chtheme/

[44] http://wammu.eu/gammu/

[45] http://freshmeat.net/projects/gtkthemeswitch/

[46] http://lists.samba.org/archive/rsync/2005-January/011472.html

[47] http://lists.debian.org/debian-qa-packages/2008/09/msg00122.html

[48] https://launchpad.net/ubuntu/+archive/primary/+files/gtk-chtheme_0.3.1.orig.tar.gz

[49] http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=500076

[50] http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=463013

[51] S. Podobry. Using PREfast for Static Code Analysis. Apriorit Inc, Published online. 11 Mar 2011. http://is.gd/MlT4N5

[52] L. Wang, J.R. Cordy and T.R. Dean, "Enhancing Security Using Legality Assertions", Proc. WCRE 2005 - IEEE 12th International Working Conference on Reverse Engineering, Pittsburgh, November 2005, pp. 35-44.