SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •
Progress of Projects Supported by NSFC

December 2012 Vol. 55 No. 12: 2785–2799
doi: 10.1007/s11432-012-4741-1

More efficient automatic repair of large-scale programs using weak recompilation

QI YuHua[1], MAO XiaoGuang[1]*, WEN YanJun[1], DAI ZiYing[1] & GU Bin[2]

[1]*School of Computer, National University of Defense Technology, Changsha* 410073*, China;*
[2]*Beijing Institute of Control Engineering, Beijing* 100190*, China*

Received June 20, 2012; accepted November 9, 2012

Abstract Automatically repairing a bug can be a time-consuming process especially for large-scale programs owing to the significant amount of time spent recompiling and reinstalling the patched program. To reduce this time overhead and speed up the repair process, in this paper we present a recompilation technique called weak recompilation. In weak recompilation, we assume that a program consists of a set of components, and for each candidate patch only the altered components are recompiled to a shared library. The original program is then dynamically updated by a function indirection mechanism. The advantage of weak recompilation is that redundant recompilation cost can be avoided, and while the reinstallation cost is completely eliminated as the original executable program is not modified at all. For maximum applicability of weak recompilation we created WAutoRepair, a scalable system for fixing bugs with high efficiency in large-scale C programs. The experiments on real bugs in widely used programs show that our repair system significantly outperforms Genprog, a well-known approach to automatic program repair. For the wireshark program containing over 2 million lines of code, WAutoRepair is over 128 times faster in terms of recompilation cost than Genprog.

Keywords weak recompilation, efficiency, automated program repair, automated debugging, experiment

Citation Qi Y H, Mao X G, Wen Y J, et al. More efficient automatic repair of large-scale programs using weak recompilation. Sci China Inf Sci, 2012, 55: 2785–2799, doi: 10.1007/s11432-012-4741-1

1 Introduction

Automatic program repair is considered to be more difficult and tedious than bug finding [1], and plays an important part on providing fully automated debugging. Recently some promising techniques [2–7] have been proposed to automate this process, reducing the cost of software maintenance. Generally, the repair process is three-phase procedure: locating the program bug (fault localization); generating the candidate patches based on some specified rules (patch generation); and validating these patches through various test cases over and over until a valid patch is found (patch validation). Most publications in the field mainly focus on the problem of how to generate candidate patches for validation and deployment. Cost reduction techniques for the process of patch validation have rarely been considered. In this paper we plan to optimize the program recompilation process in the patch validation phase, and thus speed up the whole repair process.

*Corresponding author (email: xgmao@nudt.edu.cn)

The ability to automatically repair bugs with high efficiency is critical for some applications. For example, it is believed that a software bug caused the failure of the Phobos-Grunt Mars probe[1)], an attempted Russian sample return mission to Phobos in 2011. Owing to the poor communication with Phobos-Grunt, it is not possible to identify the bug with the probe, let alone fix it manually. Automatically repairing the defective software, under the existing conditions, may be the last resort to rescue the probe. Furthermore, the time to repair the bug is limited owing to the finite energy power of Phobos-Grunt. And hence, the lesser time required to successfully repair the bug, the greater the chance of making the probe work normally. In this case, the failure of Phobos-Grunt would most probably have been avoided if the bug was able to be automatically repaired with high efficiency.

However, correcting a large-scale program using current approaches can be a time-consuming activity. In automatic repair, several data sets that cause program failure as well as a regression data set need to be identified. The auto repair environment applies multiple candidate patches (in terms of mutations) to the code, and the patch that allows the failure traces to execute while still satisfying the regression set is accepted as the solution to the bug. When a candidate patch is generated, the target program after application of the patch has to be recompiled and reinstalled to check whether the patch is valid [8]. However, finding a valid patch is often an exercise in trial and error. Moreover, with so many patches to validate for a large-scale program, recompiling each of the patched programs results in high compilation cost, making the repair process less efficient. For instance, the time required to fix a bug in the php[2)] system is, on average, 1.84 hours [9]; recompiling the patched php system takes, on average, 78.4% of the whole repair time [2]. Owing to the high compilation cost, most current approaches for automated program repair are less feasible for large-scale programs.

Existing approaches for automated program repair [4,9], both positive test cases encoding normal behaviors and negative test case exposing the fault are tested to validate the effectiveness of each candidate patch. The goal is to obtain a valid patch satisfying all test cases. Consider a candidate patch: First apply the patch to the source code of the target program. Then utilize a traditional compiler like gcc to recompile the source code to a new executable program. Finally, run all test cases to validate the patch. We note that with these approaches all the files associated with the modified code fragment, which could mean all the source files in extreme cases, are recompiled, while in some case, the patched program has to be reinstalled to ensure that everything works correctly. This kind of recompilation technique that modifies or regenerates the executable program is referred to as strong recompilation.

Although strong recompilation redundancy is evident in most cases, it becomes a much more serious problem in large-scale programs [10]. Since programmers are usually competent, the programs developed by them come very close to the correct versions [11]. Furthermore, we found that all 55 bugs in our experiments could be fixed in one function having analysed the programs with Genprog [9]. That is, at least half of the existing bugs (55 of the 105) can be fixed in one function, since the benchmark programs and defects "are indicative of real-world usage" [9]. Thus, it is reasonable to assume that many faults are merely simple ones, and can be fixed by small syntactical changes, which is also the foundation of mutation testing [12] and Genprog. Since recompiling each patched program completely with the strong recompilation technique suffers from redundant recompilation cost, reducing the redundant cost can make the repair approaches more efficient and scalable.

Although there are already some existing techniques, such as incremental compilation [13] and the compiler-integrated technique [14], which are used by mutation testing [12] to reduce the recompilation cost, these techniques, in essence, still belong to strong recompilation for the simple reason that target executable programs are modified or regenerated for each recompilation. For strong recompilation, however, a subsequent reinstall is necessary in some cases, because some executable programs are invisible unless they are explicitly installed. For example, in the Linux system the command "make install" has to be executed (after recompilation) to ensure that the makefile-based imagemagick and libtiff programs work normally. Unfortunately, for large-scale programs reinstallation can be expensive and often takes several seconds to reinstall with every program change.

1) http://www.spaceflightnow.com/news/n1202/06phobosgrunt/
2) Refer to the php interpreter implementation

In this paper we presents a recompilation technique called weak recompilation that aims to speed up the recompilation process, and creates a repair system integrating weak recompilation. Suppose that P is the original program comprising a set of components $C = \{c_1, c_2, \ldots, c_n\}$, such that $B \subset C$ is a subset of C, and that there is a patch pt representing the change transformation that can be applied to C to produce C'. Consider P' to be the patched P containing the components B' with pt applied to B. In weak recompilation, instead of recompiling all the source files dependent on the modified code fragment, we recompile only the altered components of B' to a shared library. Then, we use function indirection, a mechanism allowing some specified functions to be replaced, to implement the update to the original program (hence, the term "weak"). The implementation of weak recompilation depends on the specification of a set of components and the language in which the target programs are written. In general, components correspond to elementary elements in the target program with a specific syntax. Functions, classes and even files are all examples of components. In this paper, we have implemented weak recompilation on programs written in C using our view of components. We refer to components as the functions included in the source code.

Compared to strong recompilation, the advantage of weak recompilation is that it requires a lower compilation cost. Specifically, the benefits are twofold. First, recompilation time for weak recompilation is clearly reduced and relatively stable. While redundancy is obvious in strong recompilation, weak recompilation, which recompiles only the modified components, does not suffer from the redundancy problem irrespective of the scale of the target program. Second, in weak compilation reinstallation time is completely eliminated regardless of the fault complexity. Of course, there is the possibility that some faults are so complex that they can only be corrected by modifying relatively more of the source code. Although this would negate the first advantage of weak recompilation, because much time would still be needed for compiling the relatively large patch file, which would scale up with the amount of original source code changed, weak recompilation still has the advantage of no reinstallation cost. Unlike strong recompilation reinstalling the recompiled program to make the patched program correctly work, we can make a patch work by loading the shared library produced in weak recompilation, and then implementing the update with the aid of the function indirection mechanism. As the original executable program is not regenerated or modified at all, subsequent reinstallation is not needed.

Based on the above discussion, in this paper we propose a scalable recompilation technique called weak recompilation, and present WAutoRepair, an automated repair system using this technique. We also present an evaluation thereof on six large-scale programs: libtiff, gmp, python, imagemagick, php, and wireshark with real-life bugs. Compared to Genprog [2], a well-known automatic repair approach, our approach requires lower recompilation cost, and thus has a higher repair efficiency.

2 Weak recompilation

Weak recompilation supports flexible program recompilation whereby only the altered code fragments are recompiled. As illustrated in Figure 1, suppose that a program P consists of a set of components $C = \{c_1, c_2, \ldots, c_i, \ldots, c_n\}$. Then, assuming that $B \subset C$ is modified to repair the program, the corresponding B' is obtained. In weak recompilation, instead of recompiling all the relevant components in C to create a new executable program as is the case in the strong recompilation used by the overwhelming majority of current repair approaches, only the B' needs to be recompiled to a shared library. Clearly, weak recompilation requires lower program compilation cost than strong recompilation.

2.1 Program components

In our current work, each $c \in C$ primarily refers to one of three types: function, class, or file. The definition of the program component depends on both the language and the accuracy of the fault localization technique used to locate the cause of a failure. For example, for C programs, the class type does not need to be considered. For C++ programs, using a file-level component is more likely to be suitable.

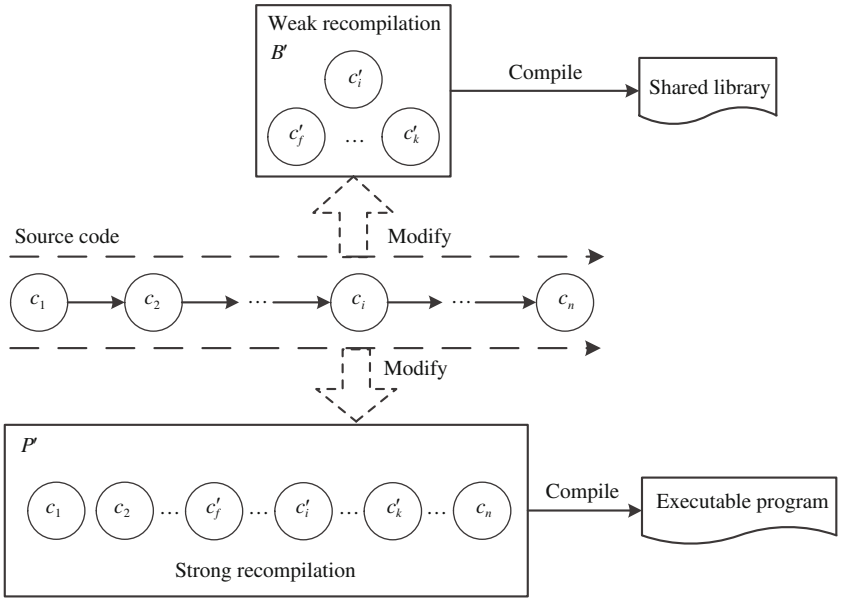

Figure 1 Weak recompilation versus strong recompilation.

In fact, function-level weak recompilation should be suitable in most cases. Many bugs are merely simply program faults that can be fixed by a few small changes for certain functions, which can be located with relatively high accuracy by existing fault localization techniques. Hence, to some extent it is highly probable that the faulty program can be successfully repaired with higher efficiency in function-level weak recompilation.

2.2 Patches

As illustrated in Figure 1, we set out to modify certain components of B in order to repair the faulty program P. In reality, in the repair process, each B' corresponding to the modified B serves as one concrete patch in terms of the shared library. This section describes the structure, automatic generation, application, and recompilation correctness of patches in weak recompilation.

2.2.1 *Patch description*

Unlike a traditional patch [15] merely describing the literal changes to the source code, a patch produced by weak recompilation includes the full code for the B' components, which will subsequently take the place of the B components to be called by the original program to check whether the patch is valid. In addition, to ensure that the patch can be correctly compiled to a shared library without errors, the generated patch also needs to include some necessary context information such as type definitions and variable declarations.

2.2.2 *Patch generation*

To ensure that the whole repair process is automatic, we have implemented a patch generator, based on CIL-1.3.7, in the OCaml language, and which has been integrated into the WAutoRepair system, to construct patches for C programs in function-level weak recompilation. Taking the source code as inputs, according to the specified rules this generator automatically produces the patch file by extracting relevant information from the altered source code. The process of patch generation is illustrated in Figure 2.

First, the intermediate files are generate from the source code. For most large-scale programs, the recompilation of one function often depends on some context information included in other files. For simplicity, for each file included in the source code we extract and write both the file itself and relevant context information to a new file called the intermediate file (*.i* file in Linux) using gcc with the flag

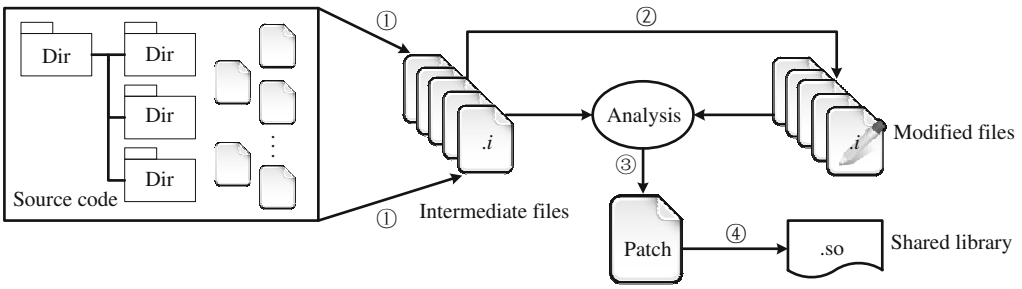

Figure 2 The process of generating a patch.

–save-temps. In fact, each intermediate file is the same as the corresponding file in the original program, but with additional context information.

Second, the source code is modified based on the intermediate files generated in the first step. Based on certain modification rules we seek to repair the faulty program by modifying only the suspicious components that have been identified by the fault localization technique.

Third, a fully automated patch is generated by analyzing the differences between the original source code and the altered source code (both based on the *.i* files). This produces the patch including only those functions whose code has been changed in the second step. We aim to compile the generated patch in the same way as for a dynamic-link library (DLL), that is, compiling the patch file to a shared library. For a DLL source code file, however, context information (e.g., type definitions and variable declarations) is included. Hence, to guarantee the correctness of the subsequent recompilation process for the patch, we need to extract some relevant context information from the original source code, and add this information to the patch. Then, the final patch is constructed from both the changed functions and context information, which subsequently assists these functions to be successfully compiled to a shared library.

Finally, using a regular compiler (such as gcc) the patch file is compiled to a dynamic patch file based on a shared library. We consider that a patch fails to compile if some error occurs during the recompilation process; this is consistent with the process of traditional strong recompilation. Note that the last three steps can be repeated or iterated to produce more candidate patches.

2.2.3 *Patch application*

Utilizing the function indirection mechanism provided by either dynamic linking (such as LD_PRELOAD in Linux) or instrumentation tools, updating of dynamic patch files can be realized. When a function is indirected, it means that calls to the function are intercepted and rerouted to another specified function. Most instrumentation tools such as Valgrind can monitor and control the execution of the target program. With the original program executed via an instrumentation tool, we can wrap suspicious functions, so that calls to these functions are intercepted and rerouted to the corresponding functions included in the shared library produced in Subsection 2.2.2.

2.2.4 *Recompilation correctness*

There exists a question of how to guarantee the correctness of the recompilation process while redundant recompilation is avoided in weak recompilation. In fact, weak recompilation is analogous to the implementation of a DLL. Both these techniques place certain functions code into a single, separate library file; then, the file is compiled to a shared library; the programs that call those functions are connected to the library at run time. The difference between these techniques is that for DLL, the library file is automatically connected when it is required, whereas for weak recompilation we implement the connection through an explicit function indirection mechanism. From the viewpoint of the user, for weak recompilation it is more likely that the patched program rather than the original one is executed. In this way, we can deliver candidate patches, in the form of shared libraries, to real, large-scale and complex programs without recompiling them completely. Hence, weak recompilation, which shares the same compiler with

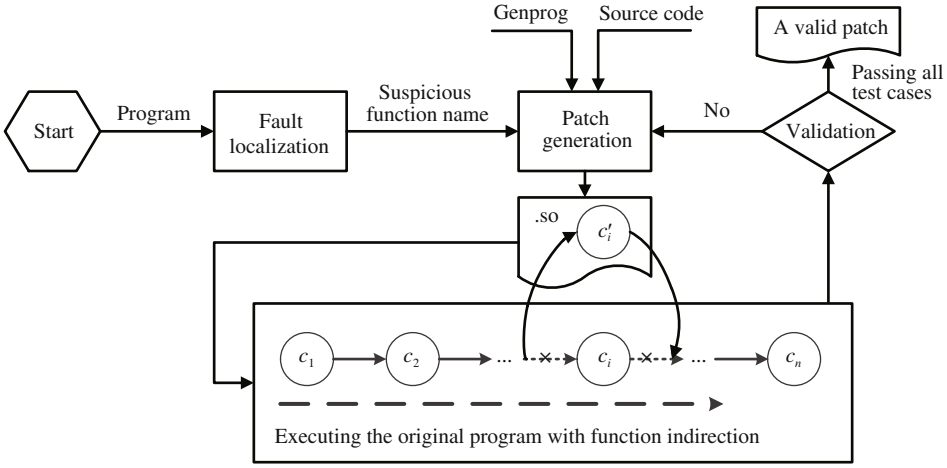

Figure 3 Framework for WAutoRepair.

strong recompilation and alters only the way of calling certain functions, does not affect either the outputs of patched functions or the recompilation correctness.

In addition, after analyzing the repair process in our experiments (see Section 4), we found that the outputs of the subject programs using both weak and strong recompilation are identical, which further validates both the correctness of the weak recompilation process and the robustness of the function indirection mechanism.

3 WAutoRepair

To justify the effectiveness of weak recompilation technique, we implemented a new repair system called WAutoRepair, which enables scalability in fixing bugs with high efficiency in large-scale C programs. The framework for WAutoRepair is presented in Figure 3. In this section, we describe this framework in detail, and then provide a case study that uses WAutoRepair to fix a real bug in the php system, which was not well repaired by the maintainers of the system.

3.1 Framework of WAutoRepair

We implemented WAutoRepair in the OCaml language. With function-level weak recompilation integrated in the framework, WAutoRepair fixes a bug in the following way: when a failure occurs, the fault information is obtained through the fault localization component; with the fault information, the patch generation component automatically generates a shared file (i.e., the patch file) according to the modification rules (i.e., adapting statements from other locations in a program) used by Genprog; with the shared library loaded, we can obtain the behavior of the patched program by executing the original program with the function indirection mechanism, and then checking whether the patched program works correctly by running it with all test cases. If the patched program satisfies all the test cases, then a valid patch has been found, otherwise WAutoRepair returns to the patch generation process and continues. WAutoRepair repeats the search process until a valid patch has been found, or when the limitations on the system have been exceeded (namely, too much time or too many attempts have elapsed). Further details of the processes illustrated in Figure 3 are given in the following subsections.

3.1.1 *Fault localization*

In our framework illustrated in Figure 3, we do not assign a specific approach for the fault localization process, since several noteworthy studies have been conducted in this area [16]. We assume that the defective areas have already been located. What needs to be done is merely to inform WAutoRepair of these areas. In the current WAutoRepair, there are two ways of achieving this: either explicitly by

specifying various suspicious function names or implicitly by providing the path information (collected by running the positive and negative test cases). In the former method, WAutoRepair tries to generate candidate patches by merely modifying these suspicious functions; in the latter case, similar to the method used in Genprog [2], WAutoRepair biases modifications toward those statements that are executed only by the negative test cases. Note that the accuracy of fault localization plays a very important role in the effectiveness and efficiency of our WAutoRepair system. In this paper, however, fault localization is not the focal point, since so much work has already been done in this area.

3.1.2 *Patch generation using Genprog*

Recall that the source code is modified according to certain modification rules (see Subsection 2.2.2). Specific to this framework, we take the rules used by Genprog to guide the modification process. Note, however, that the rules that can be utilized by WAutoRepair are not limited to those from Genprog; WAutoRepair also works well with other rules [4,15].

Genprog, a generic method for automatic software repair, can repair defects in deployed, legacy C programs without formal specifications [2,9]. Under the hypothesis that certain important functionalities omitted by a program may occur in another position in the program, Genprog can repair a program by copying a few lines of code from other parts of the program or modifying the existing code. Genprog takes advantage of a modified version of generic programming to maintain a population of variants that are represented by corresponding pairs consisting of the abstract syntax tree (AST) and the weighted path. Specifically, using a stochastic process, a candidate patch is generated using one of the generic algorithm operations: mutation or crossover. The variant is merely the AST representing the program with the patch applied. Then, to validate this patch, test cases are required: first, the variant is compiled, and its fitness is evaluated by executing test cases. If some patch satisfies all the test cases, Genprog terminates the process of searching and declares that a valid patch has been found.

As an innovative method, Genprog has successfully fixed many bugs existing in several off-the-shelf programs. However, Genprog suffers from the scalability problem; that is, for large-scale programs, the time required for recompiling and reinstalling a patched program increases rapidly, making the repair process less efficient.

In consideration of the scalability problem, our framework merely borrows the above modification rules to guide the way patches are generated. Once a patch has been generated, a new method described in the following section, is used to validate the patch rather than using Genprog's method.

3.1.3 *Executing the program with function indirection*

As the patched program is not completely recompiled, we obtain the execution behaviors of the patched program by executing the original program with function indirection, which provides a mechanism for wrapping certain specified functions. Specifically, there are two ways of implementing the function indirection mechanism provided by WAutoRepair: instrumentation and dynamic linking.

For instrumentation, the mechanism of function indirection is widely supported by many tools such as Valgrind and Pin. WAutoRepair uses Valgrind[3)] as its instrumentation tool. Valgrind can instrument and monitor both C and C++ programs. In addition, compared to other instrumentation tools, indirection specifications and semantics for Valgrind to wrap functions are relatively simpler. This was one important reason for our selection of Valgrind. However, instrumentation tools often slow down the execution of the target program to some extent [17], making the execution of each test case more time-consuming. For example, programs executed with Valgrind run roughly 5 times slower than normal. Fortunately, in most cases, this extra cost is less than the full recompilation overhead.

For dynamic linking, to defer much of the linking process until the program starts running, WAutoRepair overrides some of the library functions with corresponding ones included in the patch file by setting the LD_PRELOAD environment variable[4)]; as an alternative function indirection mechanism, dynamic

3) http://valgrind.org/downloads/current.html
4) http://lca2009.linux.org.au/slides/172.pdf

linking has a lower cost than instrumentation for indirect functions. However, dynamic linking with LD_PRELOAD, which requires target programs to be dynamically compiled, is not generalized for all C programs since some C program such as some of the php modules are statically compiled. Hence, if the target program is dynamically compiled, dynamic linking is used to implement the function indirection mechanism; otherwise Valgrind is used as the instrumentation tool to indirect functions.

3.1.4 *Validation*

As in some of the related work [4,9,18,19], there is no a priori guarantee that the generated patches will in fact fix the bugs. Hence, when a candidate patch is produced by the patch generation process, we need to check whether the patch is valid. Consider a patch file based on a shared library, which is produced by the patch generation process. The validation process runs the patched program (see Subsection 3.1.3) against both the set of positive test cases T_P, characterizing the required behavior of the program, and the set of negative test cases T_N, encoding the fault to be repaired. WAutoRepair considers the patch to be valid only if the patched program successfully satisfies all the test cases in $T_P \cup T_N$, otherwise WAutoRepair returns to the patch generation process and continues the search process until the set limits have been exceeded.

In practice, it is feasible that too many test cases may need to be executed to guarantee that the obtained patch is indeed valid with a high confidence level. Hence, WAutoRepair reduces the execution overhead by applying a random sampling technique [9], which first evaluates the patched program on a random sample of test cases before evaluating all the test cases. If the preliminary patch passes the random sample of test cases, WAutoRepair evaluates the patch on all test cases as is the case in strong recompilation to guarantee that it is really a valid patch.

3.2 Case study

The php program, which provided an excellent interpreter for the web-application scripting language, is popular and widely-used. However, a bug[5], that is, an incorrect null value check, exists in version 5.2.1.

Although a bug report was submitted, we found that the bug was not suitably repaired, and can still be reproduced in most recent version (php-5.4.3). Using WAutoRepair, the bug was successfully fixed, and the fix report is publicly available[6].

To fix the bug, WAutoRepair first had to obtain information about the defective code fragment causing the fault. In this case, we searched the error messages thrown by running php against the negative test from the source code (e.g., the "grep" command in Linux), and found that the error messages arose from the function *_php_ibase_bind* included in *ibase_query.c*. Hence, the *_php_ibase_bind* function was considered to be the suspicious function in the sense that the bug was most likely located in this function. Note that some other fault localization techniques with higher accuracy, such as Tarantula [20] or even debugging experiences, were also used in conjunction with WAutoRepair (see Subsection 3.1.1).

Using the name of the suspicious function and source files, the patch generation process first analyzed the source code to produce the intermediate files. Then one patch was generated at a time by modifying the body code of *_php_ibase_bind* according to the modification rules provided by Genprog. An example patch is shown in Figure 4(b). Next, we describe the details of the patch.

As shown in Figure 4(b), the candidate patch consists mainly of two parts. The first is a macro called I_WRAP_SONAME_FNNAME_ZU (line 3) from the included valgrind.h (line 1). This is an encoded name that Valgrind[7] recognizes when reading the symbol table information. By this way, the calls to *_php_ibase_bind* are automatically intercepted and rerouted to the function encoded by the macro in the loaded library if the program is executed with Valgrind. The other part is the function body (starting at line 4), which is actually executed in the subsequent validation process. Specific to this example, the function body is just the original body with the candidate patch applied. In addition, to ensure that the

5) https://bugs.php.net/bug.php?id=54426
6) https://bugs.php.net/bug.php?id=62300
7) Note that, in this case, WAutoRepair implements function indirection by way of instrumentation, because php is statically compiled

```
  …
1 static int _php_ibase_bind(XSQLDA *sqlda, zval ...
      ibase_query *ib_query TSRMLS_DC)
  {
  …
2      case IS_NULL:
      /* complain if this field doesn't allow NULL values */
3     if (! (var->sqltype & 1)) {
4         _php_ibase_module_error("Parameter %d:
          non-empty value required" TSRMLS_CC, i+1)
5             rv = FAILURE;
6     } else  {
7         buf[i].sqlind = −1;
              }
8     if (var->sqltype & SQL_ARRAY) ++array_cnt;
9     continue;
  …
  }
…
```

(a)

```
1 #include "/usr/local/include/valgrind/valgrind.h"
2 #include </home/grace/WAutoRepair/php/ibase_query.c>
3 int I_WRAP_SONAME_FNNAME_ZU(NONE,_php_ibase_bind)
  (XSQLDA *sqlda , zval ***b_vars , BIND_BUF *buf ,
        ibase_query *ib_query)
  {
  …
4         case IS_NULL:
  /* complain if this field doesn't allow NULL values */
5 //if (! (var->sqltype & 1)) {
6   //_php_ibase_module_error("Parameter %d:
      // non-empty value required" TSRMLS_CC, i+1)
7         // rv = FAILURE;
8 //} else {
9   buf[i].sqlind = −1;
          //}
10 if (var->sqltype & SQL_ARRAY) ++array_cnt;
11 //continue;
  …
  }
```

(b)

Figure 4 Php case study. (a) Code fragment for _php_ibase_bind; (b) corresponding wrapper.c file.

file can be successfully compiled to a dynamic link library, line 2 includes some relevant context information required by the function body.

Once a candidate patch has been produced, WAutoRepair immediately compiles it to a shared library with the file name wrapper.so. Then, with wrapper.so loaded, WAutoRepair runs the original php with Valgrind against test cases in the validation process. In this way, the candidate patch stored in wrapper.so is applied to the php program. For the purposes of this example, one negative test case reproducing the fault and three positive test cases (these three random tests were first evaluated prior to the evaluation of all the 4986 test cases) encoding the required behaviors were used to validate the candidate patch. Note that by increasing the number of test cases we can guarantee with a higher confidence level that a patch is actually valid. If the patch satisfies the negative test case and also satisfies all the positive test cases, WAutoRepair considers the patch to be valid and terminates the repair process, otherwise WAutoRepair returns to the patch generation process and continues the search until certain pre-defined limits have been exceeded.

WAutoRepair successfully fixed this bug, requiring on average, only 150.790 seconds to obtain a preliminary patch that satisfied the three random test cases, and 284.346 seconds to guarantee that the patch was really valid by running it against all 4986 test cases. Detailed performance results for WAutoRepair are given in the next section.

4 Experimental evaluation

This section systematically evaluates and analyzes the experimental results of using WAutoRepair to repair six representative programs with their sizes (expressed as lines of source code) differing by orders of magnitude. To demonstrate the advantages of WAutoRepair, we also show the experimental results for the comparison between WAutoRepair and Genprog in the same experimental context.

4.1 Experimental setup

We selected those programs used in most of the recent work [9] on Genprog as the experimental benchmark programs[8], excluding gzip, fbc, and lighttpd because we had difficulty reproducing their bugs. To

8) https://church.cs.virginia.edu/genprog/archive/genprog-105-bugs-tarballs/

Table 1 Descriptive statistics of benchmark programs

Program	LOC	Test cases	Version	Function indirection
libtiff	77000	78	libtiff-bug-0860361d-1ba75257	Dynamic linking
gmp	145000	146	gmp-bug-14166-14167	Dynamic linking
python	407000	303	python-bug-69783-69784	Dynamic linking
imagemagick	450000	143	imagemagick-6.5.2	Dynamic linking
php	1046000	4986	php-5.3.6	Instrumentation (Valgrind)
wireshark	2814000	63	wireshark-bug-37112-37111	Instrumentation (Valgrind)

eliminate the possible bias of benchmarks used by Genprog, we also included the imagemagick and another php bugs in our benchmark suite. All six programs are written in the C language, and widely used throughout the world. We ran WAutoRepair to fix one bug in each program; further details of these programs are given in Table 1.

For each faulty program, we first tracked down the suspicious function of each faulty programs using existing fault localization techniques, and then selected one negative test case and three positive test cases for random testing, and executed WAutoRepair to repair them in the same way as discussed in Subsection 3.2. If a candidate patch passed the random testing, WAutoRepair evaluated the patch on all the test cases in the same way as strong recompilation to ensure that it was a valid patch. Note that the suspicious functions in the first four programs in Table 1 were indirected using dynamic linking. WAutoRepair terminated when a valid patch was found, or when the predetermined limits (more than 100 trials or 3 hours computation time) had been exceeded.

For the purpose of comparison, all the experimental parameters for Genprog (including the Genprog rules used by both WAutoRepair and the original Genprog) in our experiment were the same as those used in [9], except for the weighted path, which comprised a sequence of <statement, weight> pairs. In our experiment the weighted path, in which the default weight 1.0 was assigned to each element, was merely the set of all the statements included in the suspicious functions. In other words, the genetic operations of mutation and crossover were constrained to operate only on the suspicious functions, which is consistent with the assumption of weak recompilation.

All the experiments ran on an Ubuntu 10.04 machine with a 2.33 GHz Intel quad-core CPU and 4 GB of memory. For each program, we performed 100 trials, and only recorded the trials leading to a successful repair.

4.2 Results

The experimental results are summarized in Table 2. Since WAutoRepair finds candidate patches according to the modification rules used by Genprog, the repair effectiveness (in terms of success rate) of WAutoRepair is the same as for Genprog. A detailed discussion on the effectiveness can be found in [9]. In this section we mainly analyze and compare the performance of WAutoRepair and Genprog aiming to address the following three questions:

- To what extent is the performance overhead reduced by WAutoRepair compared to Genprog?
- What determines the effectiveness of weak recompilation?
- Is weak recompilation necessary for automatic program repair?

4.2.1 *Performance analysis*

In this paper, we primarily measured the performance overhead in terms of time spent repairing each bug. As shown in Table 2, the "Validating one candidate patch" column reports the average time overhead for validating a single candidate patch, in terms of "Recompilation"[9] (the average time spent on compilation for each successful trial), "Positive tests" and "Negative tests" (the average time taken for the execution

9) For simplicity, we refer to both the recompilation and reinstallation costs as the recompilation cost, since, in reality, reinstallation arises from recompilation

Table 2 Experimental results

Program	Approach	Validating one candidate patch				Total search (s)	Check (s)
		Recompilation (s)	Positive tests (s)	Negative tests (s)	Sum (s)		
libtiff	WAutoRepair	0.257	0.447	0.161	0.865	9.430	16.173
	Genprog	13.488	0.729	0.522	14.739	126.908	
gmp	WAutoRepair	0.084	1.961	2.108	4.153	109.342	87.405
	Genprog	2.883	1.610	2.321	6.814	176.333	
python	WAutoRepair	0.295	1.481	1.454	3.230	51.314	149.631
	Genprog	14.221	1.833	1.456	17.510	256.535	
imagemagick	WAutoRepair	0.704	1.972	0.662	3.338	48.684	7.432
	Genprog	62.085	2.187	0.738	65.010	838.782	
php	WAutoRepair	0.567	4.645	2.267	7.479	150.790	284.346
	Genprog	19.937	0.205	1.004	21.146	387.790	
wireshark	WAutoRepair	0.222	2.131	0.559	2.912	8.210	39.716
	Genprog	28.622	0.891	0.941	30.454	51.695	

of test cases in a successful trial), and "Sum" (the total time overhead to validate one patch for each repair). The "Total search" column gives the entire time overhead for finding a preliminary patch that satisfies the random sampling test cases. Finally, the "Check" column gives the validation time for evaluating the preliminary patch on all the test cases. In our experiments, we found that the overwhelming majority of preliminary patches are in fact valid patches that pass all test cases, which justifies the use of the sampling technique. This is not surprising because most programs are well designed, and the changed code has only a small probability of affecting other code areas.

Clearly, with much smaller values in the "Total search" column, WAutoRepair performs better than Genprog, especially for the wireshark program containing over two million lines of code; the time that WAutoRepair requires to obtain a preliminary patch is only 15% of that needed by Genprog. Moreover, Genprog spent on average over 128 times more time compiling a single candidate patch than WAutoRepair. In addition, with the application of weak recompilation, WAutoRepair performs stably with less compilation cost regardless of the scale of the target programs. For example, even with the sizes differing by orders of magnitude, the time spent compiling one candidate patch for all six programs is less than 1 second.

4.2.2 *Effectiveness of weak recompilation*

As shown in Table 2, our weak recompilation is very effective in reducing compilation overhead, and relatively stable no matter what the scale of the target program is. So, what determines the effectiveness of weak recompilation? More specifically, what are the conditions under which WAutoRepair has the greatest advantage over Genprog?

Intuitively, for traditional strong recompilation, the recompilation overhead correlates highly with both the scale of the target program and the number of files dependent on the defective code. The larger the scale and the number are, the greater the recompilation overhead is likely to be. However, for weak recompilation the recompilation overhead is weakly correlated with the above two factors, because only a few code fragments (in terms of function-level components) are, in fact, recompiled. According to Table 2, none of the recompilation costs for WAutoRepair are greater than 1 second. As a result, it is reasonable to assume that the larger the program is or the more files there are dependent on the defective code, the more effective is weak recompilation.

4.2.3 *Necessity of weak recompilation*

The results in Table 2 show the evidence that weak recompilation is very effective in reducing recompilation overhead, although Table 2 shows that Genprog takes only a few minutes to finish its execution.

In some cases, the superior effectiveness of weak recompilation is crucial.

First, for time-critical applications, the less time required to successfully repair the bug, the greater is the chance of making the application work normally. Take, for example, the programs shipped with the Phobos-Grunt Mars probe (see Section 1). Even a delay of only a few minutes to repair the bug could make the probe deviate siginificantly from the designed orbit, reducing the chance of rescuing the probe.

Second, in some cases, Genprog spends much more time on successfully repairing a bug. For example, in Table 2 from [9], shows that it took Genprog over 6 hours to repair a bug in the fbc program. In general, Genprog needs many more trials to search for a valid patch, if the accuracy of fault localization is imprecise (meaning more suspicious statements that are irrelevant to the bug) or the bug is complex enough to require several changes in the source code. In these cases, with the same size trials, WAutoRepair applying weak recompilation requires much less time to find a valid patch compared with Genprog.

In fact, weak recompilation is most effective at reducing the recompilation overhead. For the wireshark program containing over 2 million lines of code, WAutoRepair is over 128 times faster in terms of recompilation cost than Genprog. According to Table 2, none of the recompilation times for WAutoRepair are higher than 1 second. Given that we plan to study more complex modification rules (which implies more trials) in our future research, the sum of the recompilation cost could be very high if weak recompilation were not applied.

4.2.4 *Summary*

In short, WAutoRepair requires less recompilation time to find a preliminary patch that passes the sampling test compared with Genprog. Moreover, most of the preliminary patches were confirmed to be valid patches in the sense that they pass all test cases, which justifies the poor dependence between different code areas. Thus, for certain time-critical applications, it is reasonable, at least partly, to generate a quick patch which means running relatively fewer but more important test cases to reduce the "Check" cost. In this sense, WAutoRepair has the advantage of requiring less time ("Total search" data in Table 2) to obtain a quick patch as the preliminary patch, compared to Genprog.

4.3 Threats to validity

The results of our experiments are subject to the following threats to validity:

First, we did not discuss the effectiveness of class-level or file-level components, since WAutoRepair can only be applied to C programs. In this paper, we mainly discuss the problem of repairing C programs, which is exactly the research language on which Genprog focuses. Furthermore, since our current version of the WAutoRepair system is based on Genprog using CIL to parse the C source code, WAutoRepair does not support other languages because CIL works only with C programs. In the future, we plan to study the effectiveness of weak recompilation with class-level components by replacing CIL with Clang, another code operation tool supporting class-type languages such as C++.

Second, file-level components were not discussed in our experiments. In fact, a file-level component is constructed from a series of function-level components for C programs. Hence, similar to the function-level components, we can compile all the changed files to a shared library, and implement the update to the original program using the function indirection mechanism in exactly the same way as in WAutoRepair. The advantage of using file-level components over function type components is that a file-level component does not require additional cost to gather context information (see Subsection 2.2.2), because this information is already included in the intermediate files. This additional cost, however, is often trivial unless there are several changes in the source code caused by bad fault localization. In addition, the file-level component requires greater compilation cost than a function-level or class-level component. In fact, as described in Subsection 2.1, since the overwhelming majority of faults successfully repaired by Genprog and WAutoRepair are not complex, the implementation of function-level components is sufficient for their repair.

Finally, like many other studies, a further issue is related to the possible poor generalization of the experimental results. Although we selected subject programs used in different application areas and

which are widely used throughout the world, chances are that the results tend to support the conclusions drawn from our experiment with some bias. We plan to minimize the experimental bias by increasing the number of subject programs in our future work.

5 Related work

5.1 Automatic program repair

Genprog: Because candidate patches are generated in WAutoRepair based on modification rules used by Genprog, the recent work by Werimer et al. [2,8,9] is most relevant to this study. They proposed Genprog, an automated approach for fixing bugs existing in legacy programs, and utilized an extended form of genetic programming to obtain a program variant. The search space is limited with the hypothesis that the important missing functionality, causing the bugs, is likely implemented in another location in the same program. Then, using the genetic operations of mutation and crossover, Genprog operates only on the defective parts to generate a candidate patched program. To verify the patch, both positive and negative test cases are selected and tested.

Evolutionary search: Arcuri [19] firstly proposed the idea of fixing software with search algorithms, and showed some preliminary experimental results on automatically fixing a defective sorting routine. In subsequent work [3], the possibility of automating the complex task of repairing programs and the limitations of this approach were discussed, and the prototype JAFF, a tool written in Java with the capability of automatically repairing programs written in a sub-set of the Java programming language, was developed. However, the experimental results in these research studies were not reported on real-world software with real bugs, because not all Java language specifications are supported by JAFF. Hence, the result of their work on practical applications is unknown.

AutoFix-E: Another noteworthy study on automatic program repair was carried out by Yi Wei et al. [4,21]. They presented a novel technique for automated fixing of programs with contracts [22]. A contract, which represents the associated specifications with software elements such as classes or approaches, provides a criterion to ensure the correctness and soundness of the proposed fix. Building on the work by Dallmeier et al. [23], AutoFix-E, a tool that can automatically generate and validate fixes for software faults, was developed. Taking as inputs an Eiffel class and test cases generated by the AutoTest tool, AutoFix-E first extracts the corresponding object states by means of Boolean queries, to obtain a fault profile by comparing the states of the executions of the positive and negative test cases. Then, a finite-state behavioral model, which is a representation of the normal behavior, is captured from the state transitions in the executions of the positive test cases. Finally, the candidate fixes are generated guided by the fault profile and behavioral model. To validate these fixes, AutoFix-E tests the patched software with all the test cases, and deems a fix to be valid if it successfully executes all the test cases with no failures.

Currently, the above three approaches are the most popular works for automatic program repair at the source code level. Although they generate candidate fixes according to different rules, there is at least one common point: to validate a fix, they have to execute the patched program with all test cases. Moreover, these approaches do not scale well with large-scale programs. To address the scalability problem, similar to the approach presented in this paper, we can apply the weak recompilation technique to these approaches to eliminate time spent on validation.

5.2 Cost reduction techniques for mutation testing

Similar to automated repair, mutation testing also needs to recompile the modified source code (a detailed account can be found in the survey in [12]). As a relatively mature testing technique for evaluating the effectiveness of a test set in terms of its ability to detect faults, there are several techniques for cost reduction research work, such as weak mutation and the compiler-integrated technique.

Weak mutation: Similar to weak recompilation, in weak mutation [24] a program P is also divided into several components, each of which refers one of the following types: variable, arithmetic expression

and relation, or boolean expression. Assume that component c_i is changed for mutation testing. Instead of executing the whole program and then checking the output as in strong mutation, in weak mutation the program is terminated immediately after the execution of component c_i. And the mutant is killed by comparing the program state of the mutant program with the original program state for the same test case. Clearly, weak mutation reduces the execution cost and improves the test effort, albeit with less test effectiveness. The reason for this is that the final output of the mutant may still be correct even with different intermediate states. Hence, a natural next step would be to reduce the execution cost of repairing a program based on the insight of this method. However, there is the risk that some valid patches may be taken for invalid ones, leading to false negatives.

Compiler-integrated technique: In mutation testing, the mutant is almost the same as the original program with only minor differences in most cases. To suppress the redundant compilation cost, DeMillo et al. [14] introduced a method to integrate support for program mutation directly into a compiler. This compiler-integrated technique works as follows: Taking the original program as inputs, the compiler produces two outputs: a target executable program and a collection of additional machine instruction sequences. Moreover, each patch, comprising one or more of these instructions, can be applied to the target executable program to yield a patched program without recompilation. Clearly, this technique can improve test efficiency by reducing the compilation cost. However, in order to use the technique the compiler has to be modified, which may lead to an incompatibility problem. In this paper, without the need for modifying the compiler, we suppress the redundant compilation cost by using our weak recompilation technique.

To the best of our knowledge there has been little research on cost reduction techniques for automatic program repair, although several studies for automatically finding and repairing program bugs do exist. The body of this work focused on the problem of compilation cost reduction in the repair process, which is even more important for large-scale programs.

6 Conclusion and future work

Current approaches for automated program repair often suffer from high compilation cost especially for large-scale programs. In this paper we presented a weak recompilation technique aiming to address to this problem. Rather than recompiling all the relevant files as is the case in popular approaches, using weak recompilation we recompile only the altered components of the target program. We also created a system called WAutoRepair, which incorporates the weak recompilation technique. The experimental results reveal that WAutoRepair is very effective in reducing the redundant compilation cost, and gracefully outperforms the original Genprog. Complete experimental results for the experiments discussed in this paper are available at http://sourceforge.net/projects/wautorepair/files/.

Owing to the relatively high efficiency of applying weak recompilation, WAutoRepair spends less time on validating a candidate patch. That is, WAutoRepair enables us to validate more candidate patches within a limited time. In the future, we plan to support more complex repair rules (meaning that more trials), such as contacts-based [22] and co-evolution-based repair rules [3], to enhance WAutoRepair's ability to repair more types of faults. In addition, we also plan to extend WAutoRepair's ability of repairing the faulty programs written in other languages, such as Java and C++. In this way, we hope WAutoRepair can be more mature and practical in assisting in automatically repairing faulty programs.

Acknowledgements

This research was supported in part by National Natural Science Foundation of China (Grant Nos. 90818024, 91118007), National High-tech R & D Program of China (Grant Nos. 2011AA010106, 2012AA011201), and Program for New Century Excellent Talents in University. We thank W. Weimer et al. for their noteworthy study on Genprog, based on which the WAutoRepair system was built.

References

1 Harman M. Automated patching techniques: the fix is in: technical perspective. Commun ACM, 2010, 53: 108
2 Le Goues C, Nguyen T, Forrest S, et al. GenProg: a generic method for automatic software repair. IEEE Trans Softw Eng, 2012, 38: 54–72
3 Arcuri A. Evolutionary repair of faulty software. Appl Soft Comput, 2011, 11: 3494–3514
4 Pei Y, Wei Y, Furia C A, et al. Code-based automated program fixing. In: International Conference on Automated Software Engineering, Lawrence, 2011. 392–395
5 Debroy V, Wong W E. Using mutation to automatically suggest fixes for faulty programs. In: International Conference on Software Testing, Verification and Validation, Paris, 2010. 65–74
6 Jin G L, Song L H, Zhang W, et al. Automated atomicity-violation fixing. In: Programming Language Design and Implementation, San Jose, 2011. 389–400
7 Xu M W, Li Q, Yang Y, et al. Self-healing routing: failure, modeling and analysis. Sci China Inf Sci, 2011, 54: 609–622
8 Weimer W, Forrest S, Le Goues C, et al. Automatic program repair with evolutionary computation. Commun ACM, 2010, 53: 109–116
9 Le Goues C, Dewey V M, Forrest S, et al. A systematic study of automated program repair: fixing 55 out of 105 bugs for $8 each. In: International Conference on Software Engineering, Zurich, 2012. 3–13
10 Byoungju C, Mathur A P. High-performance mutation testing. J Syst Soft, 1993, 20: 135–152
11 DeMillo R, Lipton R J, Sayward F G. Hints on test data selection: help for the practicing programmer. Computer, 1978, 11: 34–41
12 Jia Y, Harman M. An analysis and survey of the development of mutation testing. IEEE Trans Softw Eng, 2011, 37: 649–678
13 Schwartz M D, Delisle N M, Begwani V S. Incremental compilation in magpie. In: SIGPLAN Symposium on Compiler Construction, Montreal, 1984. 122–131
14 Edward W K J. Compiler-Integrated software testing. Ph.D. thesis. West Lafayette: Purdue University, 1991
15 Ackling T, Alexander B, Grunert I. Evolving patches for software repair. In: Genetic and Evolutionary Computation, Dublin, 2011. 1427–1434
16 Ali S, Andrews J H, Dhandapani T, et al. Evaluating the accuracy of fault localization techniques. In: International Conference on Automated Software Engineering, Auckland, 2009. 76–87
17 Nethercote N. Dynamic binary analysis and instrumentation. Ph.D. thesis. Cambridge: University of Cambridge, 2004
18 Le Goues C, Weimer W, Forrest S. Representations and operators for improving evolutionary software repair. In: Genetic and Evolutionary Computation, Philadelphia, 2012. 959–966
19 Arcuri A. On the automation of fixing software bugs. In: International Conference on Software Engineering, Leipzig, 2008. 1003–1006
20 Jones J A, Harrold M J. Empirical evaluation of the tarantula automatic fault-localization technique. In: International Conference on Automated Software Engineering, Long Beach, 2005. 273–282
21 Wei Y, Pei Y, Furia C A, et al. Automated fixing of programs with contracts. In: International Symposium on Software Testing and Analysis, Trento, 2010. 61–72
22 Wei Y, Furia C A, Kazmin N, et al. Inferring better contracts. In: International Conference on Software Engineering, Waikiki, 2011. 191–200
23 Dallmeier V, Zeller A, Meyer B. Generating fixes from object behavior anomalies. In: International Conference on Automated Software Engineering, Auckland, 2009. 550–554
24 Howden W. Weak mutation testing and completeness of test sets. IEEE Trans Softw Eng, 1982, 8: 371–379