Evaluation of Diverse Compiling for Software-Fault Detection

Andrea Höller, Nermin Kajtazovic, Tobias Rauter, Kay Römer, and Christian Kreiner
Institute for Technical Informatics, Graz University of Technology
{andrea.hoeller, nermin.kajtazovic, tobias.rauter, roemer, christian.kreiner}@tugraz.at

***Abstract*—Although software fault prevention techniques improve continually, faults remain in every complex software system. Thus safety-critical embedded systems need mechanisms to tolerate software faults. Typically, these systems use static redundancy to detect hardware faults during operation. However, the reliability of a redundant system not only depends on the reliability of each version, but also on the dissimilarity between them. Thus, researchers have investigated ways to automatically add cost-efficient diversity to software to increase the efficiency of redundancy strategies. One of these automated software diversification methods is diverse compiling, which exploits the diversity introduced by different compilers and different optimization flags. Today, diverse compiling is used to improve the hardware fault tolerance and to avoid common defects from compilers.**

However, in this paper we show that diverse compiling also enhances the software fault tolerance by increasing the chance of finding defects in the source code of the executed software during runtime. More precisely, the memory is organized differently, when using different compilers and compiler flags. This enhances the chance of detecting memory-related software bugs, such as missing memory initialization, during runtime. Here we experimentally quantify the efficiency of diverse compiling for software fault tolerance and we show that diverse compiling can help to detect up to about 70% of memory-related software bugs.

I. Introduction

A reliable and safe operation of software is of significant importance in safety-critical systems, whose failures could result in loss of life, significant property damage, or damage to the environment. Recently, these software systems have increased dramatically in scope and complexity. Consequently, it becomes increasingly difficult to guarantee that the software is safe. Despite ongoing improvements in fault prevention techniques, faults remain in every complex software system. For example, to ensure the quality of software, exhaustive program testing is strongly recommended. However, program testing can only show the presence of bugs, but never shows their absence. Thus, it is impossible to fully test and verify that a software is fault-free. To face this problem some safety standards recommend the adoption of software fault tolerance mechanisms in order to maintain a safe operation even in the presence of faults [1].

However, establishing software fault tolerance presents some unique challenges when compared to traditional hardware fault tolerance principles. The usual way to handle hardware faults is to use redundant hardware. In accordance with hardware redundancy, multiple program replicas implementing the same logical function are executed in multiple hardware channels and a voting scheme can detect a fault if the outputs of the replicas differ. In contrast to operational hardware faults, software faults (i.e., bugs in the source code) exist in every instance of the software. A consequence of a fault can be that the faulty unit stops and no bad output is produced. Such fail-stop faults can be detected relatively easily by typical fault tolerance techniques, such as a watchdog. However, there are also Byzantine faults, which are harder to detect. These are faults where the system continues to run but produces incorrect outputs. Thus, to protect against Byzantine faults, a simple addition of redundancy or traditional monitoring techniques are not enough - diversity is needed. The goal of diversity methods is to increase the probability that if components fail, that they fail on disjoint subsets of the input space, such that they lead to different consequences and can be detected [2].

A cost-efficient automated software diversification method is to exploit differences of off-the-shelf compilers. Intuitively, the diverse usage of compilers enhances the chance of detecting programmatic faults of the compiler. Furthermore, compilers use a wide range of optimization levels to improve the performance, such as speculative branch prediction and the production of branch-free code. These optimizations enable diverse behaviors, even if the same source code base is used. Previous research has concentrated on diverse compiling to increase robustness against hardware faults. However, so far little attention has been paid to the evaluation of whether compiler diversity can also be exploited to enhance the software-fault tolerance as shown in Fig.1. This paper contributes a step towards filling this gap by showing that diverse compiling helps detecting memory-related Byzantine faults, which are particularly hard to prevent and to detect. In this paper, we

- present benchmarks that represent memory-related software bugs to
- evaluate the impact of different compilation variants on the number of faults leading to crashes, and to
- quantify the effectiveness of diverse compiling.

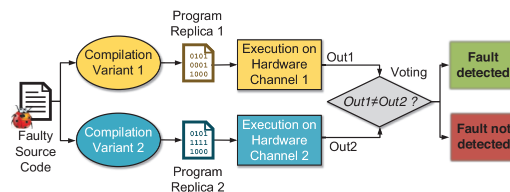

Fig. 1. Principle of diverse compiling for software-fault tolerance. The software bug is detected, if the outputs of the diverse variants do not match.

978-3-9815370-4-8/DATE15/©2015 EDAA

II. Background and Related Work

A. Terminology

With respect to persistence, a fault can be permanent or transient and according to the phase of creation or occurrence, there is a distinction between development faults and operational faults [3]. While development faults are introduced either during software or hardware development, operational faults denote hardware faults that occur during operation. In this work, we focus on permanent faults that are introduced during software development. In this case a fault and a resulting error are both commonly called bug.

Software-fault prevention techniques aim to prevent the origin of these software faults during development [2]. Therefore testing, model checking, bug finding tools and reviews are used. Additionally, fault tolerance is used to prevent that an existing fault leads to a failure in the system. Fault tolerance consists of two phases: fault detection and system recovery. Fault handling techniques, such as rollback and rollforward, can be applied after a fault is detected. However, in this paper, we only deal with fault detection.

B. Redundancy and Software-Fault Tolerance Techniques

Typical redundancy techniques are *M-out-of-N* (MooN) architectures, where *M* channels out of a total *N* channels have to work correctly. In this paper, we focus on the *1oo2* architecture meaning that there are two redundant channels, which are compared by a voter. If the outputs do not match, an error is detected. Consequently, the system can react to this fault depending to the application. For example, it can go into a safe state to prevent serious hazards. Since the voter is a single point of failure, it has to be highly reliable. Thus, it has to guarantee a high level of integrity (i.e., by being certified with a high integrity level as described in safety standards). A *1oo2* system can guarantee data integrity as long as only one channel fails. However, if both variants are identical and include the same systematic fault, they fail in the same way and the fault is not detected. Thus diversity is needed to make it possible for the same fault to lead to different consequences. A classic approach to add diversity is *N*-version programming. This means that based on the same specification, several development teams work independently to design and implement *N* versions. Since this approach is very cost-intensive, much research in recent years has focused on the automated introduction of software diversity [4]. One of these techniques is diverse compiling.

While diversity techniques try to cope with any kind of unpredictable design faults in software, exception mechanisms handle predictable problems, such as insufficient resources [5]. In this paper, we only focus on diversity as a way of achieving fault tolerance. However, it should be used complementary to exception mechanisms.

C. Influence of Compilers on System Dependability

Several studies have demonstrated that the tolerance against random hardware faults depends on the compilation. For example, the authors of [6] and [7] have shown that compiler optimizations affect the architectural vulnerability towards soft errors. Furthermore, approaches of how to change the compilation process in order to automatically harden the robustness against hardware faults have been presented [8], [9]. However, there remains the need to investigate the effects of compiling on software-fault tolerance.

D. Compilation-based Diversity Techniques

Previous research has suggested methods to automatically synthesize diversity in software [4]. Diverse compiling is such a technique. As summarized in Table I several goals are targeted in the literature about diverse compiling.

TABLE I. The goals of diverse compiling research

Approach	*Security*	*Fault Tolerance*		
		Hardware faults	*Software faults*	
			Compiler	*Executed SW*
Wheeler et al. [10]	x			
Larsen et al. [11]	x			
Lovric et al. [12]		x		
Gaiswinkler et al.[13]		x		
DO-178B standard [14]			x	
Shing et al. [15]			x	
This work				**x**

Recently, automated diversity techniques have gained much attention in the security domain. The idea is to improve the security of a system by introducing uncertainty in the target. This should increase the cost to attackers by randomizing programs, which forces attackers to target each system individually. How compilers can be used to increase the security by introducing diversity has been shown in [10] and [11]. A promising approach is the multicompiler, which introduces randomness in the binary during compilation [16]. We think that basic ideas of this research could be adapted to introduce diversity targeting the improvement of the fault tolerance.

The use of diverse compiling for hardware fault tolerance has already been proposed in 1994 [12]. In [13] it has been shown that it can be used to detect register faults.

The dependability of a compiler is crucial for the safety of a software system, because it directly affects the final executable. Thus, safety standards, such as the DO178B standard [14], state that the development of a compiler should meet the prescribed safety process. However, it also mentions that the tool qualification process may be adjusted if multiple compilers are used and the dissimilarity of these compilers can be ensured [17]. This approach has also been used in [15].

In [17] it is mentioned that compiler diversity can help to find bugs in the software during runtime. However, the authors provide no information about which kind of bugs can be detected or how efficient diverse compiling is for software-fault tolerance. This paper attempts to contribute a step towards filling this gap by presenting a quantitative evaluation of the efficiency of diverse compiling regarding software-fault tolerance.

III. Software-Fault Tolerance Through Diverse Compiling Approach

We propose to use diverse compilation to increase the fault tolerance regarding faults in the source code of the executed software. A large class of typical software bugs are faults in the memory management [18]. They can be caused, for example, by buffer overflows and uninitialized reads. These

faults can be hard to handle, since the resulting corrupted memory locations can make it difficult to link the failure to the root cause. For example, during program execution, one corrupted memory location can propagate errors to other memory locations. Diverse compiling can help in the detection of such bugs, since different compilation variants often arrange the memory in different ways. Optimizations organize the memory layout in such a way that the execution is accelerated or that the memory consumption is kept low. For example, the GCC flag `-fmerge-constants` attempts to merge identical constants across compilation units [19]. Now consider a bug that causes a buffer overflow corrupting the value of a variable, which does not immediately lead to any failure. After some time, this variable is used as an address pointer, which points at one of those address locations, where the unoptimizing compiler would place the redundant constants. When reading the content of the address the variable points to, the contents of these addresses are different when executing the unoptimized and the optimized program variant.

To summarize, diverse compiling enhances the tolerance regarding memory-related faults, since the different memory management strategies increase the chance that an erroneous read operation on the same memory location returns different values. One reason for this is that the same pointers can point to different memory locations. Additionally, it is possible that a previous write operation to an erroneous address targeted the current pointer location in one variant and another address in the other variant.

In safety critical applications, an industrial practice is not to allow compiler optimizations. One reason for this is that they can introduce non deterministic behaviors. We do not propose to change this approach. However, in addition to the traditional design using non-optimized code a redundant replica could be compiled with optimizations in order to increase the level of diversity. However, especially when developing real-time systems, the influence of the method on the execution time has to be considered. When comparing the outputs of the redundant replicas the voter has to wait until the slowest version finishes. This increases the worst case execution time to the runtime of the slowest variant in addition to the checking overhead. However, a detailed timing analysis of the diverse compiling approach is beyond the scope of this paper.

IV. Evaluation Methodology and Infrastructure

A. Benchmarking

To assess the efficiency of fault tolerance mechanisms, a software including faults is required. We proceeded as follows to create benchmarks representing typical software faults:

- First, we used the well-known MiBench benchmark suite [20] as a starting point in order to represent typical automotive and telecommunication applications.
- Then, we injected the most frequent types of software faults found in field studies.
- Finally, in order to further increase the representativeness of the tested faults, we filtered out those faults that should be detected through software testing as described below.

1) Software-Fault Injection: In order to represent realistic faults, we used the SAFE software-fault injection tool presented in [21]. The injection of software faults is relatively new compared to the extensive research on hardware fault injection. The approach is to deliberately introduce faults into a software in order to assess its behavior in the presence of faults. This technique is recommended in several safety standards [1]. The faults are injected by small changes in the program code. Various versions of a program are created, where each version includes another fault as exemplified in Fig.2. This technique is similar to the well-known mutation test technique. However, while the goal of mutation testing is to evaluate the test suite, software-fault injection is meant to assess fault tolerance techniques.

The main purpose of software-fault injection is to represent residual faults. These are those faults that are not detected by rigorous design and testing and affect the safe operation of the system. For the development of the SAFE tool several field data were analyzed to characterize faults that can realistically occur in complex software [22]. According to data collected from deployed software, the majority of bugs belong to a relatively small set of fault types and are independent from the particular system. More precisely, the SAFE tool injects a set of fault types that represent a total of about 68% faults that where collected from real-world software. Table III shows the faults injected from the SAFE tool in our examples.

The SAFE tool assures that the source code including the injected fault is syntactically correct. In order to better reproduce the fault types observed in practice, additional constraints are applied to select fault locations. For instance, the MFC fault type only affects function calls that do not return any value or do not use the return value. Another example is that an *if*-construct is only removed (MIFS) in case the construct contains less than six statements. A programmer would likely notice a missing larger *if*-construct.

2) Filtering of Faults: In order to further increase the representativeness of the tested faults, we do not consider faults

TABLE III. Overview of Injected Fault Types [22]

Fault Type	*Description*
MFC	Missing function call
MIFS	Missing IF construct and statements
MVAE	Missing variable assignment using an expression
MVAV	Missing variable assignment using a value
MVIV	Missing variable initialization using a value
WPFV	Wrong variable used in parameter of function call

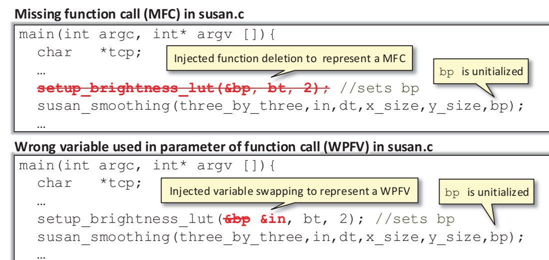

Fig. 2. Examples of injected memory-related Mandelbugs that are not detected by considered fault prevention mechanisms.

TABLE II. OVERVIEW OF GENERATED TEST CASES

Benchmark			*Number of injected fault types*							*Fault prevention*			*Bug classification*	
Name	*Description*	*LOC*	*MFC*	*MIFS*	*MVAE*	*MVAV*	*MVIV*	*WPFV*	***Total***	*Warning*	*Static A.*	***Und.***	*Bohrb.*	***Mandelb.***
Basicmath	Mathem. calc.	581	36	1	12	19	7	173	**248**	41	135	**72**	72	**0**
FFT	Fourier transform.	447	23	5	4	-	7	78	**128**	123	0	**5**	5	**0**
Susan*	Image recognition	2122	56	41	83	226	42	712	**1160**	163	0	**997**	943	**54**
GSM	GSM encoding	6116	51	53	65	0	21	352	**542**	51	62	**426**	184	**242**

* Combination of the benchmarks Susan edges, Susan corners and Susan smoothing

that result in a warning of the compiler (option flags `-Wall -Wextra`). Furthermore, we filter out those faults that are found using the Clang static source code analyzer [23]. This drastically reduces the number of test cases (see Table II). For example, 96% of the injected faults in the FFT benchmark cause compiler warnings and 54% of the faults injected in the basicmath program are detected with static analysis.

The difficulty of detecting a software fault using testing depends on the determinism of its caused failure [21]. If a fault causes an error that always leads to the same failure for a given input, the fault can be detected relatively easily during testing. These kind of bug are called Bohrbugs. Another kind of bugs are Mandelbugs, which are much more difficult to detect, since they do not always lead to an error. Typically, the occurrence of Mandelbugs depends on timing (e.g. race conditions) or on the use of resources (e.g. addressing wrong memory regions). In this paper, we consider Mandelbugs that are memory-related. To also study timing-related software bugs will be part of our future work. As shown in Table II this classification further reduces the number of evaluated test cases. Since no faults introduced in the basicmath and FFT applications meet our requirements, we focused on the Susan and GSM benchmarks. To sum up, we generated 296 faults that are particularly difficult to detect using testing for our evaluation.

B. Processor Architectures and Compilation Variants

We evaluated two different processor architectures: the Intel x86 (64-bit) architecture, representing standard desktop computing systems and the ARM926ej-s (32-bit), which is widely used in embedded applications. The Intel binaries were executed natively with Ubuntu running on a PC featuring a single-core Intel Xeon CPU (@3.1 GHz). To automatically execute ARM binaries, we used the QEMU emulator [24].

To keep the development costs low, we focus on unchanged open-source compilers that are widely used: the Gnu Compiler Collection (GCC) v4.8.2 and the LLVM Clang compiler v4.3 [25]. Both use machine-independent compiler optimizations, such as common sub expression elimination and constant merging. Additionally, they perform optimizations based on the machine description, such as instruction combining and instruction scheduling. They support flags to adjust the optimization level. We used both compilers without any optimizations (`-O0`) with performance optimizations at different levels (`-O1`, `-O2`, `-O3`) and with memory consumption optimizations (`-Os`). Note, although in general Clang attempts to copy the command-line options from GCC, there are different optimization passes in GCC and LLVM, since specific optimization flags are not shared. We evaluated all different possible combinations of the GCC and Clang compiler using different optimization flags. However, due to space limitations, we only present the most promising combinations in this paper.

C. Procedure of an Experiment

We used the benchmarks as described above and generated executables, where each version included one injected fault. These programs were tested processing several different input workloads.

The system we want to represent consists of two redundant homogeneous hardware channels executing the same software base. The only difference between the channels is introduced during the compilation phase. The output of the channels are compared by a voter, which detects a fault, if the outputs differ. Therefore, we organized an experiment as follows. We compile the same source code that includes a bug with various compiler and optimization flags. Then the resulting binaries are executed under the exact same environmental conditions (i.e., initial memory content). Finally, the different effects of the same fault are compared considering the following consequences.

- There is no influence on the execution and the output is identical to the output of the bug-free version (I).
- It delays the output (i.e., an infinite loop) which can be detected by a time-out-watchdog (T).
- It results in a segmentation fault detected by the operating system (S).
- The execution produces an incorrect output. We compare the different output versions and enumerate n different incorrect output variants as D_1,D_2,...,D_n.

While we classify time-out-faults (T) and segmentation faults (S) as fail-stop faults, faults causing erroneous outputs (D) represent Byzantine faults.

V. EXPERIMENTAL RESULTS

A. Example Output

To exemplify the generated outputs, Table IV outlines an extraction of test cases and shows if the injected faults are detected when combining chosen compilation variants. Throughout this paper the abbreviations *GOx* and *COx* represent the GCC and Clang compiler using the optimization flag *Ox*. We distinguish three different scenarios, when comparing two outputs. The first one is that the fault leads to a crash in both cases, which can be detected by monitoring mechanisms (*C*). In this case the diverse compilation does not improve the fault tolerance, since other mechanisms detect the fault. Next, we consider scenarios, where the fault manifests itself equally in both variants and leads to the same wrong value (*x*). Consequently, the fault is not detected. Finally, we consider the

TABLE IV. EXTRACTION OF RESULTS WHEN APPLYING THE SUSAN BENCHMARKS

Fault Type	x86 GCC O0	O1	O2	O3	Os	x86 Clang O0	O1	O2	O3	Os	GO1/GO3	CO0/CO3	GO3/CO3	GO0/CO3	GO3/CO0	GO1/CO3
MFC	T	T	T	T	T	D_1	T	D_2	D_2	D_2	C	✓	✓	✓	✓	C
MFC	D_1	D_1	D_1	D_1	D_1	D_1	D_1	D_1	D_1	D_1						
MFC	T	T	T	T	T	T	T	T	T	T	C	C	C	C	C	C
MVAE	I	I	I	I	I	S	I	I	I	I		✓				✓
MVAE	D_1	T	T	T	T	T	T	D_2	D_2	T	C	✓	✓	✓	✓	C
MVAE	T	T	T	T	T	T	D_1	D_1	D_1	D_1	C	✓	✓	✓	✓	C
MVAE	D_1	D_2	D_1	D_3	D_1	D_1	D_4	D_5	D_5	D_6	✓	✓	✓	✓	✓	✓
MVAE	I	I	I	I	I	D_1	D_1	D_1	D_1	D_1			✓	✓	✓	✓
MVAE	D_1	D_1	D_1	D_1	D_1	D_1	D_2	D_3	D_3	T		✓	✓	✓	✓	
MVAV	D_1	D_2	D_1	D_1	T	D_3	D_1	D_1	D_1	D_1	✓	✓			✓	✓

Fault Type	ARM GCC O0	O1	O2	O3	Os	ARM Clang O0	O1	O2	O3	Os	GO1/GO3	CO0/CO3	GO3/CO3	GO0/CO3	GO3/CO0	GO1/CO3
MFC	T	T	T	T	T	D_1	T	D_2	D_2	D_2	C	C	✓	✓	✓	✓
MFC	D_1	D_1	D_1	D_1	D_1	D_1	D_1	D_1	D_1	D_1						
MFC	D_1	T	T	T	T	T	T	T	T	T	C	C	C	✓	C	C
MVAE	I	D_1	D_1	D_1	D_1	D_2	S	D_3	D_3	D_3		✓	✓	✓	✓	C
MVAE	T	D_1	T	T	D_1	D_2	D_1	T	T	D_1	✓		C	C	✓	✓
MVAE	D_1	T	T	T	T	T	D_2	D_2	D_2	D_2	C	✓	✓	✓	C	✓
MVAE	D_1	D_2	D_3	D_3	D_4	D_5	D_6	D_6	D_6	D_6	✓	✓	✓	✓	✓	✓
MVAE	D_1	D_2	D_2	D_2	D_2	D_1	D_1	D_1	D_1	D_1		✓	✓		✓	✓
MVAE	D_1	D_1	D_1	D_1	D_1	D_1	D_1	T	T	T			✓	✓	✓	✓
MVAV	D_1	T	T	D_1	D_1	D_2	D_1	D_1	D_1	D_1	✓	✓			✓	✓

Fig. 3. Compiler-dependability of the amount of injected faults leading to a fail-stop behavior (timeout or segmentation fault) of the Susan benchmark executing ten different input values. The deviation of the maximal and minimal number of crashes illustrates the input-dependability.

Fig. 4. Average detection coverage when applying diverse compiling with selected compiler options. The small bars indicate the maximal and minimal number of detected faults for different tested inputs.

scenarios, where the diverse compiling causes different impacts of the same fault (✓). These kinds of faults are detected due to diverse compiling. This can be the case, if the fault leads to a crash in one variant and to an incorrect output in the other variant. If only using the second variant, the fault would not have been detected. This is also the case if both variants produce an output, but the outputs do not match. For example, if one channel produces a wrong value D_1 and the second produces another wrong value D_2 the voter indicates a fault. Table IV shows clearly that diverse compilation frequently causes different manifestations of the same fault.

B. Fail-stop vs. Byzantine Faults

We observed that whether a memory-related fault causes an application to crash or to deliver corrupted data, depends highly on the compilation variant and the processor architecture. This is illustrated in Fig.3 showing how many injected faults result in fail-stop failures (timeouts or segmentation faults). These faults can be detected with typical fault tolerance techniques (i.e., a watchdog). Additionally, we observed that the processed input highly influences whether a fault causes a crash. This can be seen by the deviation between the maximal and minimal number of crashes shown in Fig.3. However, the number of crashes also highly depends on the application. Whereas on average about 35% of all injected faults in the Susan benchmark result in a crash, this proportion is only less than 1% when applying the GSM benchmarks.

C. Fault Detection Coverage

Our experimental results indicate that many of the injected memory-related software faults can be detected with diverse compiling. An extract of our results showing promising compiler combinations is summarized in Fig.4 using a coverage metric defined as

$$coverage = \frac{\#detected\ faults}{\#injected\ faults}. \quad (1)$$

Table V provides more detailed information about the number of injected fault types per application and how many of these faults can be found with selected compiler combinations. The number of detected faults depends on the application and the architecture. The approach delivers better results for the GSM application than for the Susan image processing. While combinations *CO0/CO3* and *GO3/CO0* provide very promising results for GSM, the best option for the Susan application is *GO0/CO0*. Additionally, the coverage varies across different input scenarios. This is illustrated in Fig.4 showing bars indicating the maximal and minimal number of detected faults for different workloads of the same application. In general, using diverse compilers with the highest optimization flag is expected only to cause small performance penalties and provides a high fault coverage. Intuitively, the most diverse combination is to use different compilers, where one compiler does not apply any optimization and the other operates on the highest optimization level (*GO0/CO3*, *GO3/CO0*). Also this option achieves satisfactory results. However, we observed that for all tested combinations the chance of detecting memory-related software bugs has been significantly increased.

VI. CONCLUSION

Safety-critical software systems are becoming ever more complex and it is not possible to guarantee that software contains no faults. Thus, there is the urgent need for effective software-fault tolerance mechanisms. Often redundant hardware is used to tolerate hardware faults. When additionally introducing diversity, systematic faults could also be detected. Previous research has shown that compiler diversity could help to avoid common defects from compilers and improves the hardware fault tolerance. However, as far as we know, this is the first work evaluating diverse compiling to help to detect bugs in the source code of the executed software during runtime.

TABLE V. SUMMARY OF RESULTS SHOWING THE COVERAGE OF SELECTED COMPILER COMBINATIONS AS DEFINED IN (1)

Benchmark				*Coverage* x86									
Name	*#Fault Inj.*	*#Inp.*	*#Test cases*	*GO0/ GO3*	*GO1/ GO3*	*CO0/ CO3*	*CO1/ CO3*	*GO0/ CO0*	*GO3/ CO3*	*GO0/ CO3*	*GO3/ CO0*	*GO1/ CO3*	*GO3/ CO1*
MFC													
Susan	5	10	50	0%	0%	70%	70%	0%	70%	70%	70%	70%	0%
GSM	18	4	72	69%	64%	89%	97%	75%	94%	100%	89%	94%	75%
Total	**23**	**14**	**122**	**41%**	**38%**	**81%**	**86%**	**44%**	**84%**	**88%**	**81%**	**84%**	**44%**
MVAE													
Susan	18	10	180	17%	11%	39%	14%	50%	56%	56%	22%	58%	50%
GSM	26	4	104	52%	33%	79%	34%	54%	43%	55%	68%	45%	54%
Total	**44**	**14**	**284**	**30%**	**19%**	**54%**	**21%**	**51%**	**51%**	**56%**	**39%**	**53%**	**51%**
MVAV													
Susan	8	10	80	63%	38%	38%	25%	50%	38%	63%	38%	50%	50%
GSM	2	4	8	25%	75%	100%	100%	100%	100%	100%	100%	100%	100%
Total	**10**	**14**	**88**	**60%**	**41%**	**44%**	**32%**	**55%**	**44%**	**66%**	**44%**	**55%**	**55%**
MIFS													
GSM	**16**	**4**	**64**	**0%**	**0%**	**0%**	**0%**	**0%**	**13%**	**13%**	**13%**	**0%**	**13%**
MVIV													
GSM	**8**	**4**	**32**	**69%**	**44%**	**88%**	**11%**	**50%**	**69%**	**63%**	**75%**	**63%**	**50%**
WPFV													
Susan	21	10	210	63%	38%	38%	25%	50%	38%	63%	38%	62%	50%
GSM	180	4	720	58%	43%	90%	36%	74%	71%	79%	85%	74%	74%
Total	**201**	**14**	**930**	**59%**	**42%**	**78%**	**34%**	**69%**	**64%**	**75%**	**74%**	**71%**	**69%**
TOTAL													
Susan	52	10	520	41%	25%	41%	26%	45%	47%	61%	36%	60%	45%
GSM	226	4	904	58%	44%	89%	41%	72%	70%	78%	83%	72%	72%
Total	**278**	**14**	**1424**	**52%**	**37%**	**71%**	**35%**	**62%**	**62%**	**72%**	**66%**	**68%**	**62%**

Benchmark	*Coverage* ARM									
Name	*GO0/ GO3*	*GO1/ GO3*	*CO0/ CO3*	*CO1/ CO3*	*GO0/ CO0*	*GO3/ CO3*	*GO0/ CO3*	*GO3/ CO0*	*GO1/ CO3*	*GO3/ CO1*
MFC										
Susan	20%	10%	80%	50%	10%	60%	60%	80%	50%	40%
GSM	72%	64%	89%	61%	83%	83%	78%	89%	75%	83%
Total	**51%**	**42%**	**85%**	**56%**	**53%**	**74%**	**71%**	**85%**	**65%**	**65%**
MVAE										
Susan	33%	14%	44%	19%	58%	78%	56%	33%	64%	47%
GSM	73%	57%	86%	43%	73%	70%	79%	86%	59%	73%
Total	**48%**	**30%**	**59%**	**28%**	**63%**	**75%**	**64%**	**52%**	**62%**	**57%**
MVAV										
Susan	31%	50%	25%	6%	56%	56%	56%	44%	75%	31%
GSM	25%	75%	100%	100%	100%	100%	100%	100%	100%	100%
Total	**30%**	**52%**	**32%**	**15%**	**60%**	**60%**	**60%**	**49%**	**77%**	**37%**
MIFS										
GSM	**19%**	**25%**	**34%**	**13%**	**31%**	**31%**	**19%**	**23%**	**31%**	**31%**
MVIV										
GSM	**75%**	**69%**	**88%**	**44%**	**75%**	**69%**	**81%**	**94%**	**63%**	**75%**
WPFV										
Susan	21%	19%	12%	2%	19%	24%	2%	21%	2%	19%
GSM	70%	87%	90%	52%	85%	84%	85%	90%	80%	85%
Total	**59%**	**72%**	**72%**	**41%**	**70%**	**70%**	**66%**	**74%**	**62%**	**70%**
TOTAL										
Susan	27%	21%	32%	13%	37%	51%	35%	34%	39%	33%
GSM	70%	82%	90%	52%	84%	82%	84%	90%	77%	84%
Total	**54%**	**60%**	**68%**	**38%**	**67%**	**71%**	**66%**	**69%**	**63%**	**65%**

We analyzed the effect of diverse compiling on the software-fault tolerance using compilers that are widely used in practice. To limit the costs of the technique we only focused on how to exploit the options provided by existing infrastructures to introduce diversity, without changing the compilers. Our experiments have shown that the success of diverse compiling highly depends on the application, the workload of the application and the processor architecture.

The results clearly indicate that diverse compiling improves the chance of detecting software programming bugs. Our results show that when combining different optimization levels up to about 70% of the injected memory-related software faults can be detected. The main finding of this study is that the software fault tolerance is highly increased with diverse compiling regardless of the exact compiler combination. This yields the conclusion that diverse compiling reduces the vulnerability to bugs in the software.

To further investigate the practicability of diverse compiling our future work includes a worst case execution time analysis. We also plan to evaluate whether diverse compiling could help to tolerate timing-related software bugs.

REFERENCES

[1] D. Cotroneo and R. Natella, “Fault injection for software certification,” *Security & Privacy, IEEE*, 2013.

[2] L. Pullum, *Software Fault Tolerance: Techniques and Implementation*, 2001.

[3] A. Avizienis and J. Laprie, “Basic concepts and taxonomy of dependable and secure computing,” *Transactions on Dependable and Secure Computing*, 2004.

[4] B. Baudry and M. Monperrus, “The Multiple Facets of Software Diversity : A Survey,” 2014.

[5] J. Preißinger, “Fault Tolerance Through Automated Diversity in the Management of Distributed Systems,” in *International MultiConference of Engineers and Computer Scientists*, 2008.

[6] T. Jones, M. O’boyle, and O. Ergin, “Evaluating the effects of compiler optimisations on AVF,” in *Workshop on interaction between compilers and computer architecture*, 2008.

[7] M. Demertzi, M. Annavaram, and M. Hall, “Analyzing the effects of compiler optimizations on application reliability,” *IEEE International Symposium on Workload Characterization*, 2011.

[8] J. Chang, G. Reis, and D. August, “Automatic Instruction-Level Software-Only Recovery,” *International Conference on Dependable Systems and Networks*, 2006.

[9] Martinez-Alvarez et al. , “Compiler-Directed Soft Error Mitigation for Embedded Systems,” *IEEE Transactions on Dependable and Secure Computing*, 2012.

[10] D. A. Wheeler, “Fully Countering Trusting Trust through Diverse Double-Compiling,” Ph.D. dissertation, 2009.

[11] S. B. M. F. Per Larsen, Andrei Homescu, “Sok: Automated software diversity,” in *Proceedings of IEEE Security & Privacy*, 2014.

[12] T. Lovric, “Systematic and design diversity Software techniques for hardware fault detection,” in *Dependable ComputingEDCC-1*, 1994.

[13] G. Gaiswinkler and A. Gerstinger, “Automated Software Diversity for Hardware Fault Detection,” in *IEEE Conference on Emerging Technologies & Factory Automation*, 2009.

[14] L. A. Johnson, “DO-178B, Software Considerations in Airborne Systems and Equipment Certification,” 1998.

[15] A. Singh, N. Sinha, and N. Agrawal, “AVATARs for Pennies : Cheap N-version Programming for Replication,” in *6th Workshop on Hot Topics in System Dependability*, 2010.

[16] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, “Profile-Guided Automated Software Diversity,” *IEEE/ACM International Symposium on Code Generation and Optimization*, 2013.

[17] C. Wei, B. A. O. Xiaohong, and Z. Tingdi, “A Study on Compiler Selection in Safety-critical Redundant System based on Airworthiness Requirement,” *Procedia Engineering*, 2011.

[18] D. Jeffrey, N. Gupta, and R. Gupta, “Identifying the Root Causes of Memory Bugs using Corrupted Memory Location Suppression,” *IEEE International Conference on Software Maintenance*, 2008.

[19] Free Software Foundation, Inc, “GCC Documentation,” 2014. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/

[20] Guthaus, M. et al., “MiBench: A Free, Commercially Representative Embedded Benchmark Suite,” in *IEEE International Workshop on Workload Characterization*, 2001.

[21] R. Natella, “Achieving Representative Faultloads in Software Fault Injection,” Ph.D. dissertation, Federico II University of Naples, 2011.

[22] R. Natella, D. Cotroneo, J. A. Duraes, and S. Henrique, “On Fault Representativeness of Software Fault Injection,” 2011.

[23] Clang Project, “Clang Static Analyzer,” 2014. [Online]. Available: http://clang-analyzer.llvm.org/

[24] F. Bellard, “QEMU, a Fast and Portable Dynamic Translator,” in *USENIX Annual Technical Conference, FREENIX Track*, 2005.

[25] The LLVM Team, “Clang: A C Language Familiy Frontend for LLVM,” 2014. [Online]. Available: http://clang.llvm.org/