Towards Dynamic Software Diversity for Resilient Redundant Embedded Systems

Andrea Höller(✉), Tobias Rauter, Johannes Iber, and Christian Kreiner

Institute for Technical Informatics, Graz University of Technology, Graz, Austria
{andrea.hoeller,tobias.rauter,johannes.iber,christian.kreiner}@tugraz.at

Abstract. Faults in embedded systems are on the rise due to shrinking hardware feature sizes, increasing software complexity, and security vulnerabilities. Since such faults cannot be completely prevented, systems have to cope with their effects. Frequently, redundancy is used to achieve fault tolerance. However, with homogeneous redundancy common-cause faults such as software bugs or hardware faults in shared resources are not tolerated - diversity is needed.

In this paper, we highlight the potential of automatically introducing diversity via dynamic software diversity techniques. Recently, these techniques have attracted attention in the security domain. Furthermore, we present the idea of using such dynamic software diversity methods to create feedback-based systems that are able to adapt the execution of the program in such a way that the consequences of faults are leveraged. Finally, we demonstrate the approach with two use cases. We show that by using address space layout randomization - a widespread technique to prevent malicious attacks - it is possible to detect memory-related software bugs during runtime. Additionally, we illustrate the idea of adaptive dynamic software diversity by showing a simple example of how to recover from common-cause faults in the address decoder via software by inserting memory gaps with adjustable size.

Keywords: Software diversity · Self-adaptive systems · Resilience · Fault tolerance · ASLR · Redundancy

1 Introduction

Since ever more functionality is integrated into electronic devices, embedded systems have to fulfill increasing demands on high computing performance and have to realize ever more features [32]. At the same time, they have to offer dependability, since they are strongly integrated into everyday objects and play a crucial role in many critical application domains such as automotive, aerospace or critical infrastructures. Dependability is a superordinate concept regrouping different attributes such as safety, reliability, availability and security [2]. At the same time, embedded systems have to cope ever often with unforeseen scenarios caused by an increasing number of faults that jeopardize the dependability. The causes for this increasing number of faults are numerous:

A. Fantechi and P. Patrizio (Eds.): SERENE 2015, LNCS 9274, pp. 16–30, 2015.
DOI: 10.1007/978-3-319-23129-7_2

- *Random hardware faults*, such as soft errors and permanent errors due to manufacturing, process variations, aging, etc. occur more an more frequently due to shrinking hardware features sizes [28,34].
- *Software faults* are increasing due to growing software complexity [32].
- *Security attacks* pose an emerging risk, since ever more systems are interconnected [14].

To prevent such faults many systematic techniques for analyzing the dependability of systems have been proposed. For example, safety standards recommend failure mode and effects analysis, failure tree analysis or fault injection experiments [23]. These techniques are mainly used to assess the dependability of systems and to design appropriate countermeasures to prevent identified possible problematic situations in the presence of failures. However, the applicability of these approaches is limited, since they require detailed knowledge about the hard- and software system.

At the same time, the increasing demands on embedded systems lead to the trend to commercial off-the-shelf (COTS) hardware [27]. While there are processors available that are designed and produced for high reliability applications, their performance typically lags behind that of COTS multi-purpose processors. Additionally, COTS processors typically come at a much lower price than their reliability-hardened counterparts [13]. However, when using COTS-based hardware platforms, there are limited possibilities to add hardware-based fault tolerance techniques and details about the hardware (e.g. netlists, RTL models etc.) are typically not available.

Furthermore, the ever increasing complexity of embedded systems significantly complicates systematic approaches, since it is hard to identify all possible internal states and faults. To handle this challenge, we propose research towards the automated introduction of diversity in redundant systems as an alternative method to manage faults. This paper contributes towards this approach by

- presenting the idea of using the basic concepts of automated software diversity techniques that have been mainly proposed in the security domain to also increase the fault tolerance regarding non-malicious common-cause faults in redundancy-based embedded systems,
- introducing the concept of adaptive automated software diversity as a feedback-based method to react to observed malfunctions in order to establish resilience, and finally
- highlighting the potential of the approach by presenting two use cases showing that by using address space layout randomization - a widespread dynamic software diversity technique to prevent security gaps - it is possible to detect memory-related software bugs, and showing that adapting the size of dynamic memory gaps is a simple way of establishing resilience regarding memory address decoder faults.

2 Background and Related Work

2.1 Dependability and Resilience

System dependability attributes have a major impact on product development and product release as well as for company brand reputation. For this document we define dependability according to [2] as an integrating concept that encompasses the following attributes:

- *Safety*: the absence of catastrophic consequences on the users and environment.
- *Reliability*: the continuity of correct service.
- *Availability*: the readiness of correct service.
- *Security*: the concurrent existence of availability for authorized users only, confidentiality, and integrity with improper meaning of unauthorized.

To maintain safety despite faults, the system has to detect the malfunctioning in order to react in such a way that hazardous events leading to catastrophic consequences (e.g. injuries of people) are prevented. For example, a safety mechanism could switch into a safe state. To establish reliability and/or availability the consequences of the fault should be mitigated such that the system can continue the operation. Thus, resilience is needed. According to [10] software resilience refers to the robustness of a software to adapt itself so as to absorb and tolerate the consequences of failures, attacks or changes within and without the system boundaries. Consequently, to make systems resilient, they have to cope with changing circumstances regardless of their root cause. In order to deal with unforeseen events the idea of software self-adaptability has received attention [6]. For example, self-healing systems autonomously detect and recover from faulty states by changing their configuration. However, so far these techniques are mainly used in complex server systems.

Methods for embedded systems to recover from an unhealthy state are still a research challenge. Although hardware faults can be bypassed with self-modifying hardware (e.g. [24,33]), this technique is not applicable for COTS hardware and only offers limited flexibility. Thus, there remains the need for sophisticated software-based methods to handle unforeseen scenarios caused by faults.

2.2 Redundancy

Redundancy is a common way to establish fault tolerance [32]. As hardware is becoming ever cheaper due to Moore's law there are increasing opportunities to establish redundancy in a cost efficient way. For example, there is a rise of multicore technology in the embedded domain [26]. This allows to take advantage of the additional resources available in order to establish redundancy at a relatively low cost.

While spatial redundancy means that multiple program replicas that implement the same logical function are executed in multiple hardware channels, temporal redundancy techniques use only one hardware channel and perform the

same execution multiple times subsequently. Then, a voting scheme is used to detect faults by comparing the outputs of the redundant executions.

Typical redundancy techniques are *M-out-of-N* (MooN) architectures, where M channels out of a total N channels have to work correctly. For example, a Dual Modular Redundancy (*1oo2*) architecture means that there are two redundant channels, which are compared by a voter. If the outputs do not match, an error is detected. Depending on the application, the system 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 using reliable hardware components and 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, we propose to automatically introduce diversity in the execution of the software.

The authors of [7] introduced a method combining adaptability and redundancy, by adapting the number N of redundant systems dynamically. Here, we propose not to adapt the number of redundant channels, but the behaviors of the channels.

2.3 Automated Software Diversity

Recently, automated diversity gained attention in the security domain as a technique of diversifying each deployed program version [25]. This forces attackers to target each system individually. A simple way of automatically introducing diversity in execution is to use different compilers and compiler options to generate multiple program versions. It has been shown that diverse compiling not only enhances the security of systems [37] but it can also increase the hardware and software fault tolerance [11,17]. In contrast to static techniques that generate multiple diverse program versions, dynamic software diversity techniques use only one binary that is deployed and introduces the diversity during operation. More details about automated software diversity research is provided in [4,25].

3 Dynamic Software Diversity Approach

Dynamic software diversity (DSD) techniques integrate randomization points in the executable. Examples of randomization points and their parameters are shown in Table 1. Then, the same program can perform diverse executions leading to the same results [4]. Diversity in execution can mean, for example, diverse performances, diverse memory locations, or diverse order of execution. Table 2 shows examples of dynamic diversity techniques. for example, data re-expression

Table 1. Examples of adjustable parameters of dynamic software diversity methods

Randomization method	Parameter
Memory gaps between objects [5]	Gap size
Changing base address of program [5]	Base address
Changing base address of libraries and stack [8]	Base address
Permutation of the order of routine calls variables [5]	Order of calls
Permutation of the order of variables [5]	Order of variables
Insertion of NOP instructions [22]	Number of NOPs
Data re-expression / data diversity	Parameter in re-expression
$(in = f(in,k), out = f^{-1}(out,k))$ [1]	algorithm (k)

Table 2. Classification of known automated dynamic diversity uses. While 'x' indicates the given goal is reported in literature, 'o' hints to possible goals for future research.

Known Use	*Level of Diversific.*			*Time of Diversific.*			*Goal*	
	Instruction	*Function*	*Program*	*Installation*	*Loading*	*Execution*	*Security*	*Reliability*
Data randomization [1]		x				x	o	x
Memory layout randomization (e.g. [5, 8])		x	x	x			x	o
Program encoding randomization (e.g. [3])	x					x	x	o
In-place diversification (e.g. [31])			x		x		x	o
Instruction location randomization (e.g. [16, 35]	x					x	x	o
Binary stirring [36]	x				x		x	o

is a well-established method to increase the fault tolerance with data diversity by transforming the original input to produce new inputs to redundant variants [1]. After execution the distortion introduced by the re-expression is removed before comparison. So a given initial data within the program failure region can be re-expressed to an input data that circumvents the faulty region [32].

However, most of the proposed DSD techniques focus on increasing the security by introducing uncertainty in the target. Today, if an attacker finds a vulnerability in a software program, he can exploit that knowledge to target all running copies of that program. Automated software diversity can decrease the software homogeneity and increase the cost to attackers by randomization implementation aspects of programs. The idea to diversify a software program each time it is deployed on a target recently gained attention in the security community [25].

The goal is to force the attacker to redesign the attack each time it is applied. Consequently, the risk of widely replicated attacks is reduced. For example, to circumvent buffer overflow attacks, used memory locations can be diversified. This forces the attacker to rewrite the attack code for each new target.

We propose to use such approaches also in redundant systems. If diverse replicas are used redundantly and their state is checked with a voting mechanism, an attacker has to find vulnerabilities in both replicas. Furthermore, we assume that it also increases the chance to detect non-malicious faults. The introduced diversity could lead to the detection of faults that have not been considered by systematic fault prevention techniques. For example, it is possible that programming bugs are detected that have not been found during the test and verification stage. Furthermore, hardware faults that appear during operation and affect all redundant executions can be identified, since the hardware is used differently by the diverse replicas. To sum it up, we propose to apply DSD techniques in redundant configurations to increase the chance to detect faults and to consequently enhance the safety.

4 Adaptive Dynamic Software Diversity Approach

A promising approach for resilience, is software that offers a reliable operation despite uncertain environments. Hardware faults, software bugs, or security exploits can be regarded as sources of uncertainty in the operation that has to be handled. For example, permanent hardware faults cannot be fixed during runtime. Thus, the software has to change the way it uses the faulty hardware such that the fault is masked. Adapting the software execution is probabilistic and does not require knowledge about the exact root cause of the fault. We propose to learn from detected anomalies and to diversify the execution with adaptive dynamic software diversity (ADSD). We define ADSD as

a method of automatically and dynamically diversifying the way that hardware and/or software components are used such that it learns from previously observed anomalies in order to increase the fault tolerance.

In [20], we provide a preliminary introduction of the idea of adaptive dynamic software diversity (ADSD) as a mean to bypass faults. Here, we provide further details of the approach and present a simple illustrative use case.

Figure 1 shows the basic structure of an ADSD system. Typically, a fault tolerant system contains the program, which performs the intended functionality of the system and a decision mechanism (DM) that monitors the program execution [32]. The DM detects anomalies, indicates alarms and decides which outputs to forward. For example, the diagnosis could be a plausibility check, a voter of a redundant system, or a self-aware technique that detects anomalies. Additionally, we propose a diversification control that creates a feedback-loop. This component manages the ADSD by collecting and analyzing data on detected anomalies obtained from the DM. The program is designed in such a way that it can be randomized during execution according to parameters that can be adjusted during runtime (see Table 1). Then, the diversification control can decide to alter

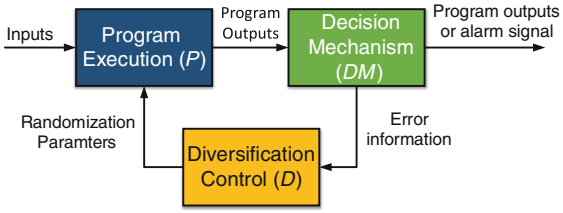

Fig. 1. Basic structure of ADSD. Based on information of a monitoring component (DM), a diversification controller decides whether and how to reconfigure the randomization mechanism of the main program [20].

the execution by changing one or multiple randomization parameters. Finally, the program reconfigures itself by using the adapted parameters.

5 Experimental Evaluation

We evaluated two use case applications in redundant configurations. We considered Dual Modular Redundancy (DMR) and Triple Modular Redundancy (TMR) for both use cases. The redundancy is established as temporal redundancy meaning that the replicas are executed one after the other. Each execution stores the corresponding result. Finally, a voter compares them after all replicas finished calculation.

A DMR system features two redundant channels. Such a system detects a fault, if the results of the redundant replicas are different. However, by purely comparing the outputs it is not possible to identify the faulty replica. This would require additional anomaly detection (i.e., plausibility checks). Here, we do not apply such mechanisms. Thus, we adapt the randomization mechanism of both replicas.

In a TMR configuration three replicas perform the same calculation. If all three outputs are different a warning is signaled and no output is forwarded, since the whole system seems to be corrupted. If at least two replicas provide the same output, this output is regarded to be correct and is forwarded. If two outputs match and one output is different, it is assumed that a fault occurred in the replica that produced the different output. A replica is also considered to contain a fault if an execution leads to a fail-stop failure. These failures can be detected relatively easily, since they lead to an observable crash of the system (e.g. segmentation fault or an infinite loop). If one replica is regarded as faulty, the randomization parameter of this replica is adapted. If it is observed that all three replicas lead to different outputs, all three replicas are reconfigured. For demonstration purposes, we change the reconfiguration parameter randomly. However, in order to achieve a more effective mechanism we plan to refine this approach to a more sophisticated logic in future work.

We evaluated how many faults can be detected by using the coverage metric

$$c_{detection} = \frac{\#detected\ faults}{\#faults}, \tag{1}$$

where $\#faults$ denotes the number of injected faults that actually have an impact on the reliability of the application, since they are not masked.

The techniques can not only be used to detect faults, but also to recover from them. To evaluate the ability to recover, we define the following metric:

$$c_{recovery} = \frac{\#bypassed\ faults}{\#faults}, \tag{2}$$

where $\#bypassed\ faults$ is the number of faults that could be mitigated with the evaluated technique.

5.1 Use Case I: Detecting Memory-Related Software Bugs via Address Space Layout Randomization

Here, we illustrate the potential of DSD techniques to increase the fault detection capabilities by evaluating the potential of the widely-established DSD technique address space layout randomization (ASLR) to detect memory-related software bugs during runtime.

Address Space Layout Randomization. ASLR randomizes base addresses to protect from security attacks [12]. It randomly arranges the starting address of the executable and the positions of the stack, heap, and libraries. To support ASLR the application has to be compiled in such a way that position independent code is generated. ASLR complicates memory corruption, code injection and code reuse attacks. For example, it is an effective countermeasure against buffer overflow attacks and it can prevent an attacker from jumping to a particular exploited function in memory. Today, ASLR is the only widely-deployed probabilistic defense against security attacks. It is supported by nearly all commonly used operating systems such as Linux (since kernel version 2.6.12), Android (since version 4.0), OS X (since version 10.5) and Microsoft Windows (since Windows Vista). However, as far as the authors of this work know, it has not been evaluated for safety or reliability purposes.

Evaluation Methodology. 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 GSM application of the MiBench benchmark suite for telecommunication applications [15] as a stating point.
- Then, we injected the most frequent types of software faults found in field studies.
- Finally, in order to further increase the representativity of the tested faults, we filtered out those faults that should be detected through software testing.

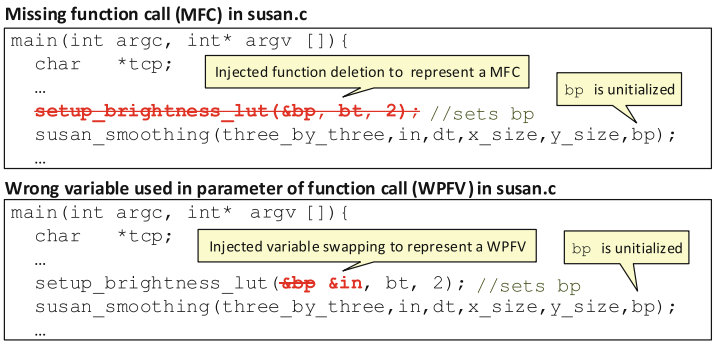

Fig. 2. Examples of injected memory-related Mandelbugs that are not detected by considered fault prevention mechanisms.

Software-Fault Injection. In order to represent realistic faults, we used the SAFE software-fault injection tool presented in [29]. It deliberately introduces faults into a software in order to assess its behavior in the presence of faults. 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 [30]. 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 % collected faults that where collected from real-world software. Table 3 shows the faults injected from the SAFE tool in our examples.

Table 3. Overview of injected fault types [29]

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

Table 4. Overview of generated test cases for the GSM benchmark

Number of injected fault types							Fault prevention			Bug classification	
MFC	MIFS	MVAE	MVAV	MVIV	WPFV	Total	Warning	Static A.	Und.	Bohrb.	Mandelb.
51	53	65	0	21	352	**542**	51	62	**426**	184	**247**

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.

Filtering of Faults. In order to further increase the representativity of the tested faults, we do not consider faults 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 [9]. This drastically reduces the number of test cases (see Table 4).

The difficulty of detecting a software fault using testing depends on the determinism of its caused failure [29]. 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. As shown in Table 4 this classification further reduces the number of evaluated test cases. To sum up, we generated 242 faults that are particularly hard to detect by testing for our evaluation.

ASLR Configurations. The binaries were generated with GCC v4.8.2 and executed natively with Ubuntu running on a PC featuring a single-core Intel Xeon CPU. In Linux the ASLR can be configured by a parameter called `randomize_va_space`. This parameter can be changed in order to set following options:

- `0`: Disable ASLR
- `1`: Randomize the base addresses of the stack, virtual dynamic shared object (VDSO) page, and the shared memory regions.
- `2`: Randomize the base addresses of the stack, virtual dynamic shared object (VDSO) page, the shared memory regions, and the data segment.

Experimental Results. We executed each binary containing one bug ten times for each ASLR configuration and stored the resulting output and applied this

Table 5. Detection coverage as defined of exemplary ASLR configurations regarding different programming mistakes leading to memory-related software-bugs.

Fault type		DMR configurations				TMR
Name	#Faults	0 vs.1	0 vs. 2	1 vs. 2	2 vs. 2	1 vs. 2 vs. 3
MFC	18	83 %	83 %	83 %	83 %	83 %
MIFS	15	0 %	0 %	0 %	0 %	0 %
MVAE	26	35 %	38 %	23 %	23 %	38 %
MVIV	8	38 %	38 %	13 %	13 %	38 %
WPFV	180	47 %	47 %	41 %	41 %	47 %
Total	**247**	**45 %**	**45 %**	**38 %**	**39 %**	**45 %**

procedure for the two input samples provided by the MiBench benchmark suite. The bugs never lead to a crash of the whole program, but always lead to a silent corruption of the output value.

Unfortunately, using ASLR it was not possible change the configuration of the execution in such a way that any of the injected software bug was mitigated and correct results were produced. However, frequently the erroneous outputs of diverse replicas were different leading to a high fault detection rate as shown in Table 5. Even if using the same configuration in a DMR configuration (2 vs. 2) on both redundant replicas in average about 39 % of all tested faults can be detected. When deactivating the ASLR feature on one channel and using it on the other channel (0 vs. 1 and 0 vs. 2) the detection coverage is even as high as 47 % for the tested application. The results also show that there is only a small gain of adding a third channel to establish a TMR system. The results indicate that ASLR is not only an effective method to prevent malicious attacks, but also enhances the fault detection of memory-related software bugs in redundant configurations. However, additional exhaustive experiments including more faults, benchmark applications, and input values, are required in order to make more reliable statements.

5.2 Use Case II: Adaption of Memory Gaps to Bypass Address Decoder Faults

Dynamic Software Diversity with Memory Gaps. To realize an automated software diversity mechanism we used random memory gaps as proposed in [5]. We arranged important variables used to control loops (`i`,`j`), storing the result (`n`), and manipulating the input in an early processing stage (`seed`) in a struct as shown in Fig. 3. Dummy variables with adjustable size are introduced in order to offer the possibility to manipulate the starting addresses of the protected variables. The size of the memory gaps represents the reconfiguration parameter that can be changed by the diversification control. Thus, if the diversification control decides to change a replica, the size of the memory gaps is adapted. We have chosen a random size between 0 and 64 bytes. Initially, the

```
int* bitcnt(int *results, int iterations, int gapsize){
  struct ProtVars {
    int dummy1[gapsize]; int  i;
    int dummy2[gapsize]; long j;
    int dummy3[gapsize]; long n;
    int dummy4[gapsize]; long seed;
  } protvars;
...
}
```

Fig. 3. Important variables are defined in a struct that introduces memory gaps between the variables with configurable size.

size of the memory gaps is configured differently for the three replicas (0 bytes, 32 bytes and 64 bytes). We plan further research to analyze the trade-off between a high degree-of freedom and additional memory requirements.

If no fault is detected, there is no need to diversify the program replicas. However, if there is a fault, the diversification control changes the size of the introduced memory gap. These is applied to all replicas when considering a DMR system or a TMR system where all three outputs are different. If only one output differs in a TMR configuration, only the erroneous replica is adapted.

Evaluation Methodology. Our exemplary use case is based on an implementation written in C and compiled for the ARM9 architecture. The application is the bitcount benchmark obtained from the MiBench benchmark suite for automotive applications [15]. The application counts the number of bits in an array of integers.

Fault Injection Framework. We simulated the benchmark application with a QEMU-based fault injection framework as presented in [18,19,21]. This framework allows to inject transient and permanent CPU and memory faults. We used it to inject permanent stuck-at address decoder faults. To model such faults the framework changes the address of the victim memory cells accordingly whenever it is accessed. This leads to accessing wrong memory locations.

Fault Injection Experiments. We injected 256 permanent stuck-at-1 and stuck-at-0 faults in the RAM addresses corresponding to the variables shown in Fig. 3. Due to masking effects, only 131 of these faults had an impact on the output values. Here, the injected fault lead to the situation that a wrong memory cell containing another value that the intended memory cell was addressed.

5.3 Experimental Results

As shown in Table 6 the homogeneous temporal redundant approach without the ADSD technique, failed to detect the introduced faults. The reason for this is that only one hardware channels was used and the fault in this hardware affected all

Table 6. Summary of the experimental results of use case II

	w/o ADSD	with adaptive memory gaps	
		DMR	TMR
Faults detected ($C_{coverage}$)	0 %	100 %	100 %
Faults tolerated ($C_{recovery}$)	0 %	82 %	94 %

executions in the same way, since they used exactly the same hardware resources. However, if the replicas use the memory slightly differently realized by a different initial configuration of the size of the memory gaps, the injected address decoder faults always lead to different consequences and thus are always detected.

Furthermore, if an error is detected the gapsizes are adjusted. For both redundant configurations - DMR and TMR - it was possible to recover from most of the detected faults. The TMR configuration performs better, since here it is possible to identify the faulty replica and thus only this replica is reconfigured. In summary, the results indicate that this adaptive technique is a simple and effective solution to recover from permanent address-decoder faults.

6 Conclusion

For many application domains of embedded systems dependability it is of utmost importance. At the same time the number of faults that may appear and the complexity of embedded systems increases dramatically, which hampers the application of traditional systematic approaches to prevent faults. Thus, methods are required that are able to detect unforeseen faults. To achieve this, we propose to use redundant systems that exhaustively integrate diversity in an automated way. In order to contribute towards this research direction, we proposed to exploit DSD techniques proposed in the security domain. The potential of this approach has been highlighted with a use case showing that using address space layout randomization allows to detect memory-related software bugs. Furthermore, we proposed to use DSD in a feedback-based configuration in order to establish resilience and exemplified this approach with a simple use case.

Although, the presented experiments indicate the potential of the proposed approach, more analysis is required in order to provide a stronger evidence. Thus, in the future, we plan to conduct more sophisticated and exhaustive evaluation of the proposed approaches. Furthermore, the actual realization of (A)DSD depends on the application and the considered fault model. Thus, we plan to develop techniques for specific complex applications and we hope to encourage further researchers to explore techniques based on the promising yet challenging idea of (A)DSD.

References

1. Ammann, P.P.E., Knight, J.C., Amman, P., Kngiht, J.: Data diversity: an approach to software fault tolerance. IEEE Trans. Comput. **37**(4), 418–425 (1988)

2. Avizienis, A., Laprie, J.C.J., Randell, B., Landwehr, C., Member, S.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secure Comput. **1**(1), 11–33 (2004)
3. Barrantes, E.G., Ackley, D.H., Forrest, S., Stefanović, D.: Randomized instruction set emulation. ACM Trans. Inf. Syst. Secur. **8**(1), 3–40 (2005)
4. Baudry, B., Monperrus, M.: The multiple facets of software diversity: recent developments in year 2000 and beyond, ArXiv e-prints (2014)
5. Bhatkar, S., DuVarney, D., Sekar, R.: Efficient techniques for comprehensive protection from memory error exploits. In: USENIX Security Symposium (2005)
6. Brun, Y., Serugendo, G.D.M., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Müller, H., Pezze, M., Shaw, M.: Software Engineering for Self-Adaptive Systems. Engineering self-adaptive systems through feedback loops. Springer, Heidelberg (2009)
7. Buys, J., De Florio, V., Blondia, C.: Towards context-aware adaptive fault tolerance in SOA applications. In: ACM International Conference on Distributed Event-Based System (2011)
8. Chew, M., Song, D.: Mitigating buffer overflows by operating system randomization. Technical Report CMUCS-02-197, Carnegie Mellon University (2002)
9. Clang Project: Clang Static Analyzer (2014)
10. Florio, V.D.: On resilient behaviors in computational systems and environments. J. Reliable Intell. Environ. **1**(1), 1–14 (2015)
11. Gaiswinkler, G., Gerstinger, A.: Automated software diversity for hardware fault detection. In: IEEE Conference on Emerging Technologies and Factory Automation (2009)
12. Giuffrida, C., Kuijsten, A., Tanenbaum, A.: Enhanced operating system security through efficient and fine-grained address space randomization. In: USENIX Conference on Security (2012)
13. Goloubeva, O., Rebaudengo, M., Reorda, M.M.S., Violante, M.: Software-Implemented Hardware Fault Tolerance. Springer, Heidelberg (2006)
14. Gubbi, J., Buyya, R., Marusic, S., Palaniswami, M.: Internet of things (IoT): a vision, architectural elements, and future directions. Future Gener. Comput. Syst. **29**, 1645–1660 (2013)
15. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: MiBench: a free commercially representative embedded benchmark suite. In: IEEE International Workshop on Workload Characterization (2001)
16. Hiser, J., Nguyen-Tuong, A., Co, M., Hall, M., Davidson, J.W.: ILR: where'd my gadgets go? In: IEEE Symposium on Security and Privacy (2012)
17. Höller, A., Kajtazovic, N., Römer, K., Kreiner, C.: Evaluation of diverse compiling for software fault tolerance. In: Design Automation and Test in Europe (2015)
18. Höller, A., Krieg, A., Rauter, T., Iber, J., Kreiner, C.: QEMU-based fault injection for a system-level analysis of software countermeasures against fault attacks. In: Euromicro Conference on Digital System Design2 (2015)
19. Höller, A., Macher, G., Rauter, T., Iber, J., Kreiner, C.: A virtual fault injection framework for reliability-aware software development. In: IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (2015)
20. Höller, A., Rauter, T., Iber, J., Kreiner, C.: Adaptive dynamic software diversity: towards feedback-based resilience. In: IEEE/IFIP International Conference on Dependable Systems and Networks - Supplementary Volume (2015)
21. Höller, A., Schönfelder, G., Kajtazovic, N., Kreiner, C.: FIES: a fault injection framework for the evaluation of self-tests for COTS-based safety-critical systems. In: IEEE Microprocessor Test and Verification Workshop (2014)

22. Homescu, A., Neisius, S., Larsen, P., Brunthaler, S., Franz, M.: Profile-guided automated software diversity. In: IEEE/ACM International Symposium on Code Generation and Optimization (2013)
23. ISO 26262: Road Vehicles - Functional Safety Standard (2009)
24. Jafri, S., Piestrak, S.J., Sentieys, O., Pillement, S.: Design of a fault-tolerant coarse-grained reconfigurable architecture : a case study. In: International Symposium on Quality Electronic Design (2010)
25. Larsen, P., Homescu, A., Brunthaler, S., Franz, M.: SoK: automated software diversity. In: IEEE Security and Privacy Magazine (2014)
26. Macher, G., Höller, A., Armengaud, E., Kreiner, C.: Automotive embedded software: migration challenges to multi-core computing platforms. In: IEEE Conference on Industrial Informatics (2015)
27. Madeira, H., Some, R.R., Moreira, F., Costa, D., Rennels, D.: Experimental evaluation of a COTS system for space applications. In: Proceedings of the 2002 International Conference on Dependable Systems and Networks, pp. 325–330 (2002)
28. Meza, J., Wu, Q., Kumar, S., Mutlu, O.: Revising memory errors in large-scale production data centers: analysis and modelling of new trends from the field. In: IEEE/IFIP International Conference on Dependable Systems and Networks (2015)
29. Natella, R.: Achieving representative faultloads in software fault injection. Ph.D. thesis. Università degli Studi di Napoli Federico II (2011)
30. Natella, R., Cotroneo, D., Duraes, J.A., Henrique, S.: On fault representativeness of software fault injection. IEEE Trans. Softw. Eng. **39**(1), 80–96 (2011)
31. Pappas, V., Polychronakis, M., Keromytis, A.D.: Smashing the gadgets: hindering return-oriented programming using in-place code randomization. In: IEEE Symposium on Security and Privacy (2012)
32. Pullum, L.: Software Fault Tolerance Techniques and Implementation. Springer, Heidelberg (2001)
33. Boesen, M.R., Pascal, S., Madsen, J.: Feasibility study of a self-healing hardware platform. In: Reconfigurable Computing: Architectures, Tools and Applications (2010)
34. Saggese, G.P., Wang, N.J., Kalbarczyk, Z.T.: An experimental study of soft errors in microprocessors. IEEE Micro **25**(6), 30–39 (2005)
35. Scott, K., Kumar, N., Velusamy, S., Childers, B., Davidson, J., Soffa, M.: Retargetable and reconfigurable software dynamic translation. In: International Symposium on Code Generation and Optimization (2003)
36. Wartell, R., Mohan, V., Hamlen, K.W., Lin, Z., Rd, W.C.: Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In: ACM Conference on Computer and Communications Security (2012)
37. Wheeler, D.A.: Fully countering trusting trust through diverse double-compiling. Ph.D. thesis. George Mason University (2009)

http://www.springer.com/978-3-319-23128-0

Software Engineering for Resilient Systems
7th International Workshop, SERENE 2015, Paris, France, September 7-8, 2015. Proceedings
Fantechi, A.; Pelliccione, P. (Eds.)
2015, IX, 145 p. 47 illus., Softcover
ISBN: 978-3-319-23128-0