J Inf Process Syst, Vol.15, No.6, pp.1438~1448, December 2019
https://doi.org/10.3745/JIPS.04.0151

ISSN 1976-913X (Print)
ISSN 2092-805X (Electronic)

JOURNAL OF INFORMATION PROCESSING SYSTEMS JIPS

Error Recovery Script of Immunity Debugger for C# .NET Applications

Rupali Shinde*, Min Choi*, and Su-Hyun Lee**

Abstract

We present a new technique, called VED (very effective debugging), for detecting and correcting division by zero errors for all types of .NET application. We use applications written in C# because C# applications are distributed through the internet and its executable format is used extensively. A tool called Immunity Debugger is used to reverse engineer executable code to get binaries of source code. With this technique, we demonstrate integer division by zero errors, the location of the error causing assembly language code, as well as error recovery done according to user preference. This technique can be extended to work for other programming languages in addition to C#. VED can work on different platforms such as Linux. This technique is simple to implement and economical because all the software used here are open source. Our aims are to simplify the maintenance process and to reduce the cost of the software development life cycle.

Keywords

Debugger, Reverse engineering, VED

1. Introduction

Nowadays, Intermediate languages such as .NET and Java are widely used among modern software development programming languages. Source code written in one of these programming languages is compiled to an intermediate language; the compiled code is utilized by the user to deploy or install the application. Intermediate code is difficult to understand yet useful for making improvements to the application. Making modifications to intermediate code is difficult due to its complexity; however, debuggers can be utilized to simplify the process of extracting and modifying binaries of intermediate code. CodePatch is the latest error recovery tool developed by MIT for three types of programming error recovery using binaries [1]. Error recovery is a major task in the maintenance phase of the software development life cycle (SDLC). Typically, the duration of software development may last 1 to 2 years, while the maintenance phase can span 5 to 10 years [2]; more time is spent on maintenance (i.e., fixing or correcting) than on actual development. Quality assurance is affected by buggy code to a significant extent; significant maintenance also results in increased software development life cycle costs.

In the field of software development, programming languages and platforms are constantly evolving;

Manuscript received December 6, 2018; first revision April 16, 2019; accepted May 8, 2019.

Corresponding Author: Su-Hyun Lee (sleepl@changwon.ac.kr)

* Dept. of Information and Communication Engineering, Chungbuk National University, Cheongju, Korea (rupaliraut1989@gmail.com, cmin1229@gmail.com)

**Dept. of Computer Engineering, Changwon National University, Changwon, Korea (sleepl@changwon.ac.kr)

software error recovery tools must be flexible to accommodate these changes. In this paper, we present division by zero error recovery using reverse engineering of code. Our main objective is to make error recovery performed during the maintenance phase flexible and easy to use. Compile time errors can be easily detected and corrected during the development phase so that we are left working with only runtime errors. Division by zero errors are examples of runtime errors; these are most often detected after product release. In this paper, we present available techniques, our method, and a sample application.

2. Related Work

Programming error detection and correction research started with the identification and classification of errors. In 1947, Richard Hamming presented Hamming codes used for error correction [3]. Significant research on source code followed and research in reverse engineering started in the early nineties. Further research is being done and is in three main domains: program analysis, design recovery, and software visualization. In 2015, Sidiroglou-Douskos et al. [1] proposed error recovery by automatic code patching using C/C++ and Python. Most research until now has been done with C, C++, and Python; however, a fair amount of research has also been done with .NET. Reverse engineering .NET applications is a tedious process due to the metadata present in the source code; this characteristic is the reason we chose a .NET application to perform the experiment. There are various techniques, listed below, that can be used for error recovery.

- Code Phage: A system for automatically transferring correct code from donor applications into recipient applications that process the same inputs to successfully eliminate errors in the recipient [1]. In this method, binary code is used. Code Phage works for out of bound access, integer overflow, and division by zero errors.
- N-Version programming: N-version programming, as the name suggests, develops multiple applications from the same specification. All implementations are executed, and the results are compared to detect for any faulty versions. The expense of N-version programming and a perception that multiple implementations may suffer from common errors and specification misinterpretation have limited the popularity of this approach [4].
- Program fracture [5]: The program fracture method only works at the code level. In this method, code is broken into multiple applications and is recombined again to implement an application with equivalent functionality.
- Machine learning and data mining: Data mining is used to suggest corrective and preventive measures to correct logical and syntactical errors [6]. In this technique database with profiling of missing invariant. A comparison of correct and buggy code is performed and the system suggests corrections.
- Tools: Tools such as source-to-source transformation [7], Kali [8], and Prophet [9] were developed to make error recovery automatic. Some tools are specific to a programming language and work only for a specific type of error. Error recovery with tools is easy to perform; however, developing a tool that performs with 100% accuracy is difficult. The cost of error detection is charged by the developer.
- CodeCarbonCopy (CCC): CCC is a system for transferring code from a donor application into a recipient application. CCC starts with functionality identified by the developer to transfer into an

insertion point (again identified by the developer) in the recipient [10].

- Logical error recovery: We have presented logical error recovery of C# applications using the software ILSpy [11]. In this method, the application is reverse engineered by using binary code. The binary code patching method is used here; in this method, error-causing code can be detected and corrected manually with the correct code [12].

3. Proposed Technique

In mathematics, division by zero is an operation where division is performed with the divisor (or denominator) equal to zero. Such an operation can be formally expressed as $a/0$. In calculus, with a/b and with b approaching zero, the limit approaching from the right is $+\infty$, and the limit approaching from the left is $-\infty$. In computer programming, the solution to this problem is to return a value of 'undefined' or to simply return an error. To overcome division by zero errors, we propose a method called VED (very effective debugging). In this method, we can automatically detect the error and take preventive measures per user preference. With our technique, we can avoid abnormal program crashes and safely execute the application without error.

3.1 Block Diagram

Fig. 1 displays a block diagram of VED. In Fig. 1, a C# application is used in a portable executable (PE) file format. CFF Explorer [13] is used to detect the format of the PE file; the most suitable debugging tool for the PE file format is then selected. Immunity Debugger [14] is most suitable for our application because it is lightweight and works well with .NET applications. We have added a Python script to the back end of the debugger to display error information. Immunity Debugger contains a Python component (COM), which helps with Python script execution. Python scripts can communicate with Immunity Debugger through binary code and the Python API. CFF Explorer and Immunity Debugger are the tools used here; both tools are open source and are easily available.

- CFF Explorer [13]: A freeware suite of tools including a PE editor called CFF Explorer and a process viewer. The PE editor has full support for PE32/64, including special field's description and modification, hex editor, import adder, signature scanner, signature manager, extension support, and disassembler.
  CFF Explorer is the first PE editor with support for .NET internal structures. The use of CFF Explorer is to get the Portable Executable file format when the application source is unknown, and the user wants to reverse engineer it with a reverse engineering tool. Many reverse engineering tools with different functionality are available; users can choose the most appropriate tool according to their specific needs. The C# application file type is called PE32 .NET Assembly, as shown in Fig. 2.
- Immunity Debugger [14]: Immunity Debugger provides a powerful new way to write exploits, analyse malware, and reverse engineer binary files. It builds on a solid user interface with function graphing, includes the industry's first heap analysis tool built specifically for heap creation, and has a large and well supported Python API for easy extensibility. The Debugger is a Python 2.7 based debugger, making it easy to interface with using hooks or scripts written in Python.

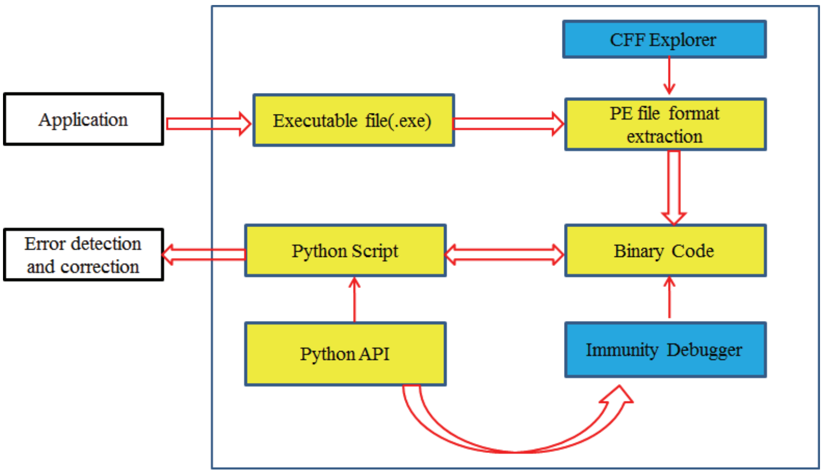
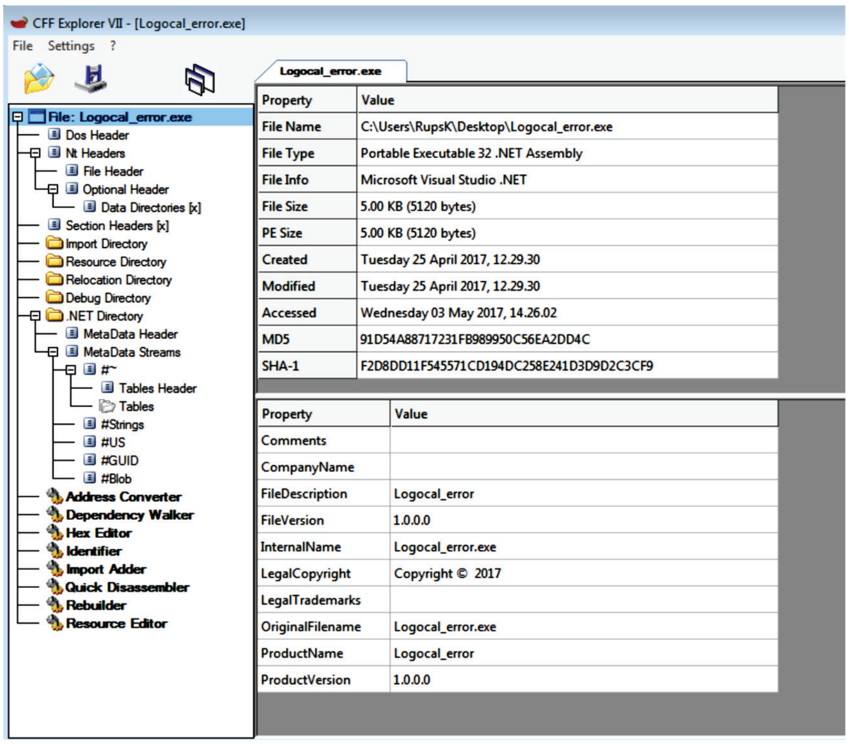


Fig. 1. Block diagram of VED.



Property	Value
File Name	C:\Users\RupsK\Desktop\Logocal_error.exe
File Type	Portable Executable 32 .NET Assembly
File Info	Microsoft Visual Studio .NET
File Size	5.00 KB (5120 bytes)
PE Size	5.00 KB (5120 bytes)
Created	Tuesday 25 April 2017, 12.29.30
Modified	Tuesday 25 April 2017, 12.29.30
Accessed	Wednesday 03 May 2017, 14.26.02
MD5	91D54A88717231FB989950C56EA2DD4C
SHA-1	F2D8DD11F545571CD194DC258E241D3D9D2C3CF9

Property	Value
Comments	
CompanyName	
FileDescription	Logocal_error
FileVersion	1.0.0.0
InternalName	Logocal_error.exe
LegalCopyright	Copyright © 2017
LegalTrademarks	
OriginalFilename	Logocal_error.exe
ProductName	Logocal_error
ProductVersion	1.0.0.0

Fig. 2. CFF Explorer.

- Immunity debugger has following key features.
  - It is specifically for security Industry
  - This Debugger can cut exploits development time by 50% means have good performance power.
  - It comes with simple interface
  - It is lightweight; gives fast debugging compare to IDE pro and OlluDBG.

- Scripting language is Python that makes scripting easy.
- Immunity Debugger's Python API includes many useful utilities and functions. Your scripts can be as integrated into the debugger as the native code.

3.2 Role of Script

Immunity Debugger is a debugger with functionality designed specifically for the security industry. It provides simple and easy-to-understand interfaces. This debugger is written in Python, and it provides a robust and powerful scripting language for automating intelligent debugging. The script can be loaded and modified during runtime. The VED script is using the inbuilt function of the debugger. Immunity is a Python based debugger, so the script is written using Python 2.7.1. A code snippet of the script is shown below.

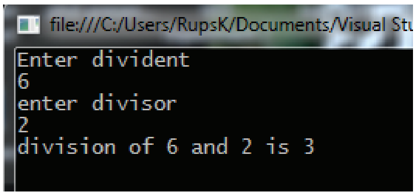
```
def main(args):
    imm=immlib.Debugger()
    evento = imm.getEvent()
    if evento:
        if isinstance(evento, ExceptionEvent):
            for a in evento.Exception:
                if a.getType = isFltDivideByZero
                    list = []
                    list.append("Restart")
                    list.append("Exit")
                    imm.comboBox("correction measure",list)
                    if "Restart":
                        imm.restartProcess(mode= -2)
                    if "Exit":
                        imm.quitDebugger(self)

                    imm.log("Exception: %s (0x%08x)" % (a.getType(), a.ExceptionCode), focus = 1)
                    imm.log("Exception address: 0x%08x" % a.ExceptionAddress)
                    imm.log("Exception num param: %d" % a.NumberParameters)
                    for value in a.ExceptionInformation:
                        imm.log(hex(value))
        else:
            imm.log("Last event type: 0x%08x (%s) " % (evento.dwDebugEventCode, str(evento) ) )
        return "Works"
    else:
        return "Cannot handle this exception"
```

Immunity Debugger's Python API includes many useful utilities and functions; scripts can be as integrated into the debugger as native code. In our experiment, the script handles divide by zero errors. Immunity Debugger has an exception event function; as soon as an exception occurs, the script determines the exception type. If a division by zero exception is determined to have occurred, debugging scripts are executed and the debugging process functions as if a script were being processed. If a script is not present, Immunity Debugger simply stops execution and shows the error in a log window (this can be difficult to understand for beginners or new users). Many scripts can be added to the system to work with different types of errors.

3.3 Conducting an Experiment with a Sample Application

The C# sample application was created, and in it two integer numbers are divided. The application takes its input from the user and shows the output as a division of two numbers. When the user inputs a non-zero value for the divisor, the application works fine; when the divisor is zero however, it throws an error and leads to the abnormal termination of the application. Fig. 3 shows the output of the sample application.

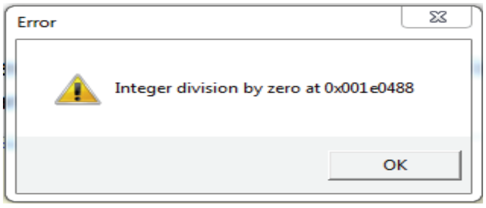
The sample application's executable file (Sample.exe) is loaded into Immunity Debugger. We have the binary code of the application displayed on the debugger window; the sample application is being debugged in Immunity Debugger. As mentioned in Section 3.2, a script is added to the root directory. As soon as the error condition is encountered during execution, the script is executed.



```
Enter divident
6
enter divisor
2
division of 6 and 2 is 3
```

Fig. 3. Sample application.

We use an event handler of class Immunity and extend its functionality. We have added the condition that if a division by zero error occurs, the script must then display an error message containing the location of the error in hexadecimal format. This script helps with error detection and with handling the abnormal termination of the application. The script output is shown in Fig. 4.

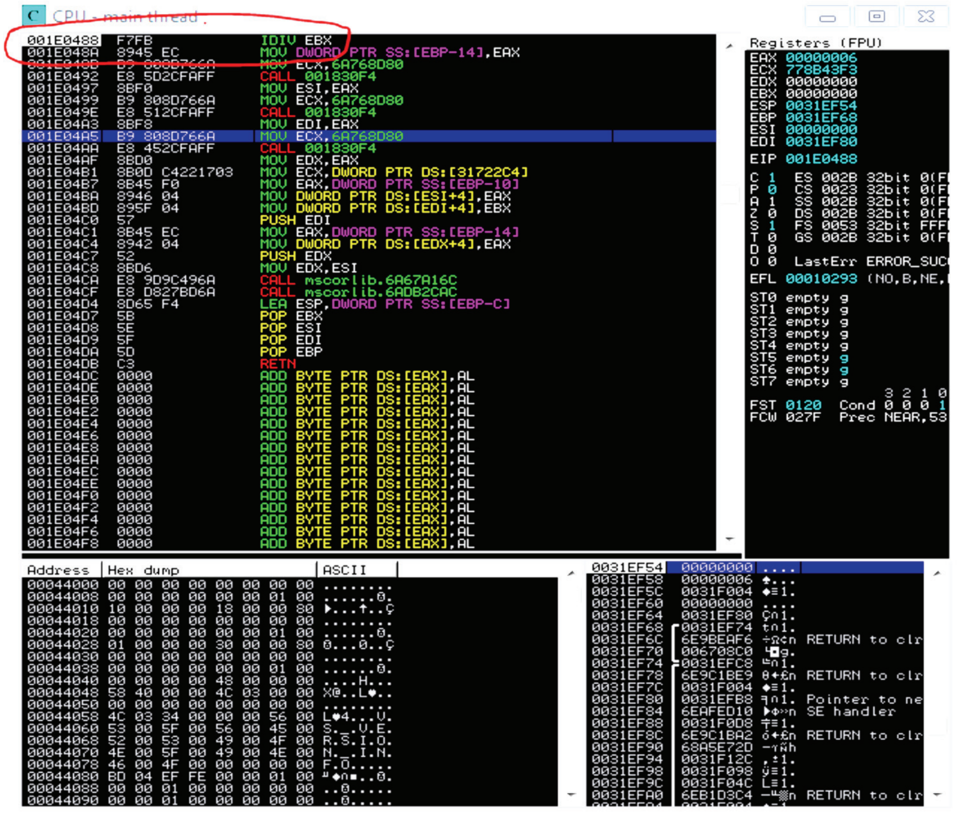


Integer division by zero at 0x001e0488

Fig. 4. Output of script.

The script output shows the error message and the location of the error. The location in hexadecimal and the instruction set casing error can be obtained by searching in the debugger window. Fig. 5 shows the output of a successfully conducted error detection experiment.

The location at 0x001E0488 is in hexadecimal format and the associated instruction is shown as IDIV EBX. Assembly instruction IDIV [15] denotes a division of two integers; EBX represents the divisor, which is present in 'B' register. A default rule of assembly language is that the accumulator in 'A' register means that the dividend is stored in 'AX' register. In the case of multiplication and division, register 'B' also works as an accumulator. After the exact working assembly code is determined, we can take preventive measures. Correcting division by zero errors is challenging and ambiguous because there is no exact mathematical solution; error prevention measures must be taken while coding. There are a few possible solutions however, as mentioned below.

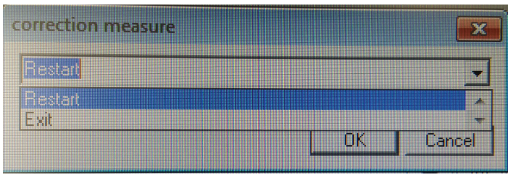


```
CPU - main thread

001E0488  F7FB           IDIV EBX
001E048A  8945 EC        MOV DWORD PTR SS:[EBP-14],EAX
001E048D  B9 808D766A    MOV ECX,6A768D80
001E0492  E8 5D2CFAFF    CALL 001830F4
001E0497  8BF0           MOV ESI,EAX
001E0499  B9 808D766A    MOV ECX,6A768D80
001E049E  E8 512CFAFF    CALL 001830F4
001E04A3  8BF8           MOV EDI,EAX
001E04A5  B9 808D766A    MOV ECX,6A768D80
001E04AA  E8 452CFAFF    CALL 001830F4
001E04AF  8BD0           MOV EDX,EAX
001E04B1  8B0D C4221703  MOV ECX,DWORD PTR DS:[31722C4]
001E04B7  8B45 F0        MOV EAX,DWORD PTR SS:[EBP-10]
001E04BA  8946 04        MOV DWORD PTR DS:[ESI+4],EAX
001E04BD  895F 04        MOV DWORD PTR DS:[EDI+4],EBX
001E04C0  57             PUSH EDI
001E04C1  8B45 EC        MOV EAX,DWORD PTR SS:[EBP-14]
001E04C4  8942 04        MOV DWORD PTR DS:[EDX+4],EAX
001E04C7  52             PUSH EDX
001E04C8  8BD6           MOV EDX,ESI
001E04CA  E8 9D9C496A    CALL mscorlib.6A67A16C
001E04CF  E8 D827BD6A    CALL mscorlib.6ADB2CAC
001E04D4  8D65 F4        LEA ESP,DWORD PTR SS:[EBP-C]
001E04D7  5B             POP EBX
001E04D8  5E             POP ESI
001E04D9  5F             POP EDI
001E04DA  5D             POP EBP
001E04DB  C3             RETN
001E04DC  0000           ADD BYTE PTR DS:[EAX],AL
001E04DE  0000           ADD BYTE PTR DS:[EAX],AL
001E04E0  0000           ADD BYTE PTR DS:[EAX],AL
001E04E2  0000           ADD BYTE PTR DS:[EAX],AL
001E04E4  0000           ADD BYTE PTR DS:[EAX],AL
001E04E6  0000           ADD BYTE PTR DS:[EAX],AL
001E04E8  0000           ADD BYTE PTR DS:[EAX],AL
001E04EA  0000           ADD BYTE PTR DS:[EAX],AL
001E04EC  0000           ADD BYTE PTR DS:[EAX],AL
001E04EE  0000           ADD BYTE PTR DS:[EAX],AL
001E04F0  0000           ADD BYTE PTR DS:[EAX],AL
001E04F2  0000           ADD BYTE PTR DS:[EAX],AL
001E04F4  0000           ADD BYTE PTR DS:[EAX],AL
001E04F6  0000           ADD BYTE PTR DS:[EAX],AL
001E04F8  0000           ADD BYTE PTR DS:[EAX],AL

Registers (FPU)
EAX 00000006
ECX 778B43F3
EDX 00000000
EBX 00000000
ESP 0031EF54
EBP 0031EF68
ESI 00000000
EDI 0031EF80

EIP 001E0488

C 1  ES 002B 32bit 0(F
P 0  CS 0023 32bit 0(F
A 1  SS 002B 32bit 0(F
Z 0  DS 002B 32bit 0(F
S 1  FS 0053 32bit FFF
T 0  GS 002B 32bit 0(F
D 0
O 0  LastErr ERROR_SUC
EFL 00010293 (NO,B,NE,

ST0 empty g
ST1 empty g
ST2 empty g
ST3 empty g
ST4 empty g
ST5 empty g
ST6 empty g
ST7 empty g
               3 2 1 0
FST 0120  Cond 0 0 0 1
FCW 027F  Prec NEAR,53

Address  Hex dump                 ASCII
00044000 00 00 00 00 00 00 00 00 ........
00044008 00 00 00 00 00 00 01 00 ......☺.
00044010 10 00 00 00 18 00 00 80 ►...↑..Ç
00044018 00 00 00 00 00 00 00 00 ........
00044020 00 00 00 00 00 00 01 00 ......☺.
00044028 01 00 00 00 30 00 00 80 ☺...0..Ç
00044030 00 00 00 00 00 00 00 00 ........
00044038 00 00 00 00 00 00 01 00 ......☺.
00044040 00 00 00 00 48 00 00 00 ....H...
00044048 58 40 00 00 4C 03 00 00 X@..L♥..
00044050 00 00 00 00 00 00 00 00 ........
00044058 4C 03 34 00 00 00 56 00 L♥4...V.
00044060 53 00 5F 00 56 00 45 00 S._.V.E.
00044068 52 00 53 00 49 00 4F 00 R.S.I.O.
00044070 4E 00 5F 00 49 00 4E 00 N._.I.N.
00044078 46 00 4F 00 00 00 00 00 F.O.....
00044080 BD 04 EF FE 00 00 01 00 ╜♦∩■..☺.
00044088 00 00 01 00 00 00 00 00 ..☺.....
00044090 00 00 01 00 00 00 00 00 ..☺.....

0031EF54   00000000  ....
0031EF58   00000006  ♠...
0031EF5C   0031F004  ♦≡1.
0031EF60   00000000  ....
0031EF64   0031EF80  Ç∩1.
0031EF68  ┌0031EF74  t∩1.
0031EF6C  │6E9BEAF6  ÷Ω¢n  RETURN to clr
0031EF70  │006708C0  └◘g.
0031EF74  ├0031EFC8  ╚∩1.
0031EF78  │6E9C1BE9  θ←£n  RETURN to clr
0031EF7C  │0031F004  ♦≡1.
0031EF80  │0031EFB8  ╕∩1.  Pointer to ne
0031EF84  │6EAFED10  ►Φ»n  SE handler
0031EF88  │0031F0D8  ╤≡1.
0031EF8C  │6E9C1BA2  ó←£n  RETURN to clr
0031EF90  │68A5E72D  -τÑh
0031EF94  │0031F12C  ,±1.
0031EF98  │0031F098  ÿ≡1.
0031EF9C  │0031F04C  L≡1.
0031EFA0  │6EB1D3C4  ─╙▒n  RETURN to clr
```

Fig. 5. Error detection.

3.4 Error Correction

Fig. 5 shows the detection of the division by zero error. In Section 3.4, we show error correction measures that are considered by our script. There are many possible options for error recovery; the selection of a particular option will vary depending on user need and choice. To implement error correction measures, we have added a combo box after the error message window. The combo box offers two options: 'Restart' and 'Exit'. Fig. 6 shows the choices presented to the user.



```
correction measure
Restart
Restart
Exit
OK   Cancel
```

Fig. 6. Error correction options displayed in the combo box.

Choice 1: "Restart"

When the user selects 'Restart', the debugger rolls back to the previous process and the user can input new values to avoid the error. This choice helps the user; the abnormal crash condition is avoided and error free debugging is enabled.

Choice 2: “Exit”

When the user selects ‘Exit’, Immunity Debugger exits the process and the debugging process stops. This option also useful for avoiding anomalous application crashes. The detailed working of VED is shown in the flow chart in Section 3.5.

3.5 Flow Chart

Fig. 7 shows a detailed flow chart of the VED technique. From Fig. 7, it is clear that if user preferences are different from the choices given in the combo box, the user can cancel out of the combo box and processed as per choice. Fig. 5 shows a cancelation button used to exit normal program execution as specified in the script. When the user chooses the cancel option, program flow of the script is halted and the user can take preventive measures with the help of breakpoint setup or code patching.

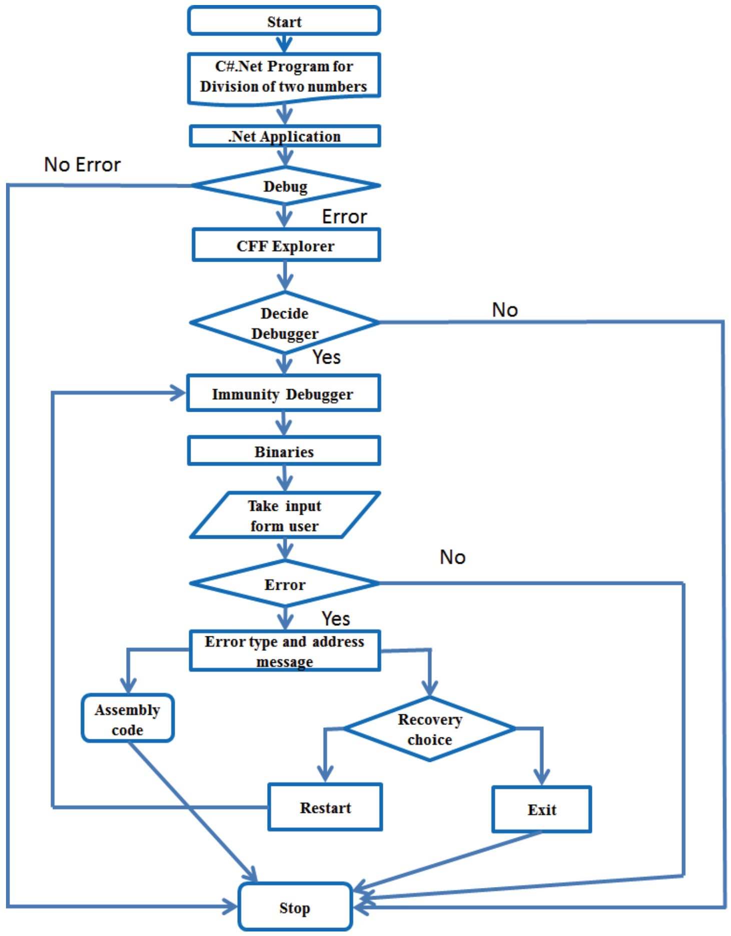


Fig. 7. Flow chart of VED.

3.6 Possible Error Correction Solutions and Challenges

In this section, we discuss other potential solutions for division by zero errors. These solutions are not included in the script; however, in the future these can be implemented to improve system performance.

We discuss each solution along with any possible challenges in implementation.

1. Breakpoint: A conditional breakpoint is set up to check if EBX=0; program execution must stop upon detection of this condition. The challenge for this solution is breakpoint setup; we have to place one every time before debugging starts and owing to the presence of metadata within a .NET application, this does not work well for .NET applications. Despite Immunity Debugger being best suited for .NET applications, breakpoints do not work well for .NET applications.
2. Replace EBX value: If we change the value of EBX from 0 to 1, program termination can be avoided; however, this leads to the wrong output being generated and the adoption of the wrong mathematical logic. This could ultimately be a significant problem if the application is for banking or scientific research.
3. Avoid Division Operation: A RET instruction can be placed to avoid division operation. This solution will enable the program to avoid termination; however, this workaround will lead to the wrong result.
4. Script: A script is implemented and an error message is displayed if necessary; the script can ask the user whether to restart or exit or roll back execution and accept another value for the divisor. This method is the most logical and practical out of all the above. Section 3.3 demonstrates this correction method, although it is not particularly useful. With this method, an abnormal crash can be avoided and the program can terminate safely. If more options are presented in the combo box, error recovery will be more efficient.

3.7 Summary

Our system presents the detection of division by zero errors and possible error correction measures. The VED experiment is still in progress. Runtime errors are mostly detected in the post-release phase and these can be recovered with this method. VED is more suitable in the post-release phase because runtime errors are mostly detected after release. All software used here is free; it is economical and easy to install. Memory requirements are very low and system performance is good; automatic detection saves time and effort. VED will be useful for all types of programming language and also for Linux operating systems. VED is sustainable; it works despite the continuous change and evolution seen in programming environments.

4. Conclusions and Future Work

Software maintenance is the most expensive part of the development lifecycle; nearly 60% to 70% of the total budget is allocated to this phase. We attempt to minimize maintenance costs and maximize customer satisfaction by providing the best software quality possible within a short time. Our method will be useful to developers by reducing the cost and effort of error recovery; this method is better suited for small scale software firms, as the tools available for performing error recovery are not free. Future work with this method is to attempt to automatically handle many types of error. Many error recovery tools are currently available in the market, but a fee that many small start-ups and software firms cannot afford is charged; VED will work perfectly for users in this market segment.

Acknowledgement

This research is supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (No. NRF-2017R1E1A1A01075128).

References

[1] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Portland, OR, 2015, pp. 43-54.

[2] A. Gupta and S. Sharma, "Software Maintenance: challenges and Issues," *International Journal of Computer Science Engineering*, vol. 4, no. 1, pp. 23-25, 2015.

[3] T. M. Thompson, *From Error-Correcting Codes through Sphere Packings to Simple Groups*. Cambridge, MA: Cambridge University Press, 1983.

[4] V. Bharathi, "N-version programming method of software fault tolerance: a critical review," *Proceedings of the 1st National Conference on Nonlinear Systems and Dynamics (NCNSD)*, Kharagpur, India, 2003, pp. 173-176.

[5] P. Amidon, E. Davis, S. Sidiroglou-Douskos, and M. Rinard, "Program fracture and recombination for efficient automatic code reuse," in *Proceedings of 2015 IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, MA, 2015, pp. 1-6.

[6] K. Deulkar, J. Kapoor, P. Gaud, and H. Gala, "A novel approach to error detection and correction of c programs using machine learning and data mining," *International Journal on Cybernetics & Informatics*, vol. 5, no. 2, pp. 31-39, 2016.

[7] Y. Khmelevsky, M. Rinard, and S. Sidiroglou-Douskos, "A source-to-source transformation tool for error fixing," in *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, Ontario, Canada, 2013, pp. 147-160.

[8] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, Baltimore, MD, 2015, pp. 24-36.

[9] F. Long and M. Rinard, "Automatic patch generation by learning correct code," *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, St. Petersburg, FL, 2016, pp. 298-312.

[10] S. Sidiroglou-Douskos, E. Lahtinen, A. Eden, F. Long, and M. Rinard, "CodeCarbonCopy," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, Paderborn, Germany, 2017, pp. 95-105.

[11] ILSpy Portable Debugger [Online]. Available: https://sourceforge.net/projects/ilspyportable/.

[12] R. K. Shinde, M. Jung, and S. H. Lee, "An experimental result of logical error recovery by reverse code engineering," in *Proceedings of the Korean Institute of Information Scientists and Engineers*, 2017, pp. 1655-1655.

[13] CFF Explorer [Online]. Available: https://ntcore.com/?page_id=388.

[14] Immunity Debugger [Online]. Available: https://www.immunityinc.com/products/debugger/.

[15] 8051 Instruction Set [Online]. Available: https://www.win.tue.nl/~aeb/comp/8051/set8051.html.

Rupali Shinde https://orcid.org/0000-0002-4631-5353

She received her B.S. in Information Technology from Mumbai University, India in 2012 and her M.S. degrees in School of Computer Engineering from Changwon National University, South Korea in 2018. Since September 2018, she is with the School of Information and Communication Engineering from Chungbuk National University as a PhD candidate. Her area of interest is Blockchain technology and programming errors recovery.

Min Choi https://orcid.org/0000-0002-8031-1022

He received his B.S. degree in Computer Science from Kwangwoon University, Korea, in 2001, and the M.S. and Ph.D. degrees in Computer Science from Korea Advanced Institute of Science and Technology (KAIST) in 2003 and 2009, respectively. From 2008 to 2010, he worked for Samsung Electronics as a Senior Engineer. Since 2011, he has been a faculty member of the Department of Information and Communication of Chungbuk National University. His current research interests include high performance computing, cloud computing, interconnection network, and embedded computing.

Su-Hyun Lee https://orcid.org/0000-0001-6966-1569

He is a professor in the Department of Computer Engineering at Changwon National University, Changwon, Korea. Dr. Lee received his B.S. in Computer Science from Kwangwoon University, Seoul, Korea, in 1987, his M.S. in Computer Science from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea, in 1989, and his Ph.D. in Computer Science from the KAIST, Daejeon, Korea, in 1994. Before joining to Changwon National University, he worked for the Electronics and Telecommunications Research Institute, Daejeon, Korea, as a member of the research staff. His research interests include programming languages, compilers, and algorithms. He is also interested in software development environments and tools, and distance e-Learning.