International Journal of Engineering & Technology, 7 (4.35) (2018) 239-243

International Journal of Engineering & Technology

Website: www.sciencepubco.com/index.php/IJET

Research paper

Preventing Stack Overflow Using Alternative Stack Approaches

Khairol Amin Mohd Salleh [1], Abdul Rahim Ahmad [2*], Roslan Ismail [3]

College of Computer Science and Information Technology
Universiti Tenaga Nasional, Kajang, Selangor, Malaysia
**Corresponding author E-mail: abdrahim@uniten.edu.my*

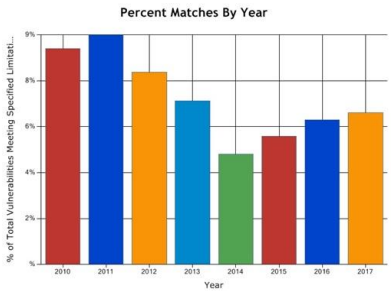
Abstract

Buffer overflow marks a phenomenon of a malicious technique employed by attackers, as reported in the NIST statistics. This paper presents a method of implementing a dual stack approach using software to protect the data stack from experiencing the attack by using 3 types of architecture, ranging from parallel program, multi-threading to a simple sequential subroutine. The current research on dual stack may require new hardware or a modified version of compiler which may complicate the implementation. These implementations spark some major issues in code backward compatibility with some changes in the language semantics especially in handling the movement of data to and from the dual stack. This paper discusses the implementation of Alternative Stack prototypes in 3 types of architecture and observation on its behavior during the performance and security test. The test has been benchmarked against the programs that are compiled with Microsoft Security Cookie. The Alternative Stack Architecture 3 prototype displayed a significant performance against the benchmarked programs whilst maintaining the confidentiality, integrity and availability of the programs.

Keywords: *Buffer overflow; stack overflow; alternative stack; software security*

1. Introduction

Buffer overflow is predominantly one of the known software vulnerability that has impacted the computer system since 90s. According to the NVD (US National Vulnerability Database) statistics provided by US National Institute of Standards and Technology (NIST), in the year 2017, buffer overflow recorded 6.25% from the total of 12,251 reported vulnerabilities as illustrated in Fig 1.



Year	Matches	Total	Percentage
2010	412	4,639	8.88%
2011	392	4,150	9.45%
2012	418	5,288	7.90%
2013	349	5,187	6.73%
2014	361	7,937	4.55%
2015	342	6,487	5.27%
2016	384	6,447	5.96%
2017	783	12,521	6.25%

Figure 1: NIST statistics on Buffer Overflow

The total buffer overflow vulnerability marks the highest value for the year 2017 with 783 vulnerabilities since 2010 and this shows that this problem is still persisting. There are a number of creative solutions that have been developed as defence against buffer overflow, especially in mitigating the buffer overflow attack that has been targeting the software stack which is known as stack overflow. Most of the solutions that protect the stack involved the need for new hardware and compiler changes which may introduce complexity in the implementation stage. This research proposed a software base approach that can be easily implemented and match the current performance of the solution which could be offered through the compiler implementation.

2. Related Works

It is interesting to note that the technique applied in defence against stack overflow can be divided into three (3) categories; buffer tagging, shadow stack and dual stack. These solutions have evolved vigorously since late 1990s till today to mitigate the different attacking techniques adopted by malicious software.

2.1 Buffer Tagging

The buffer tagging technique indicates that the area between data buffer and the stack address storage will be marked with special characters such as carriage return, line feed, null,-1, as implemented by Cowen [2] in Stackguard. Microsoft chooses a random number method that will be assigned during loading time and XOR it with the Return Address. The latest version of Stackguard provides option for developers to choose which method will be used for buffer tagging. The idea behind this technique is that each time the function returns to its caller, it will ensure that no buffer has been leaked by checking the existence of the special characters. If the special characters are no longer available or have been overwritten, this indicates that the stack overflow has taken place, and the program will be flagged for termination [2][5][6][8].

The integrity check code for the special characters is added at the prologue and epilogue of each function during compiling. Piromsopa [8] marked every word area with "secure bit" if the address falls into the same segment through a hardware implementation, leaving the external address unmarked. This is done on the pretext that every external address has a malicious intention. Once the integrity check has detected missing secure bit, the program will be signalled for termination.

One of the challenges for this technique is that the attacker managed to imitate (replicate) the special characters or guessed the

random number in order to bypass the integrity check at the epilogue[4]. The new Stackguard [2] and Microsoft Security Cookie [15] could overcome this weakness by XOR'ing the random number with the return address which happens only during runtime in the prologue. As for the securebit implementation, marking every word will eventually slow down the application performance.

2.2 Shadow Stack

The Shadow Stack technique requires storing of return address in a safe place during the start of every function, and the backup copy or return address will be compared [2][5][6] or overwritten before returning to the caller. The storage area can be a protected memory area within or outside the program memory segment [11][14], or placing a special hardware [1][3][9][13] to store the backup copy of the return address. T. Chiueh [13] uses the Linux mprotect system call to secure the backup copy of the return address.

The additional code for the return address integrity check will be added during the compiling of the program in the prologue and epilogue of the function which reflects the same approach as in the buffer tagging method. There are some solutions in adopting dynamic insertion of additional prologue and epilogue codes during execution which is advantageous for program that has no source code. Saravanan Sinnadurai [10] adopted the binary rewriting technique which performs insertion of integrity check code during execution by using a binary rewriting framework software known as DynamoRIO. This approach encountered some performance issue since each instruction call needs to go through the binary rewriting framework. The binary framework will then call the user defined function to insert the prologue and epilogue in runtime mode. INTEL [6] proposed the integrity test checking codes should be embedded in the processor itself through its CET (Control-Flow Enforcement Technology) proposal.

Before giving control to the caller routine, the return address will be compared and if the address does not match, the program will be flagged for termination. Vendicator [14] avoided the integrity check and used the backup return address to overwrite the original stack address location to enhance performance.

One of the common challenges in this technique is to measure how fast can the prologue and epilogue manage the backup return address and where to store the return address in a safe location. The issue of Frequent termination of program could also contribute to Denial of service attack as in the buffer tagging technique.

2.3 Dual Stack

The 3rd classification of defence will be addressing the problem pertaining to the legendary stack anatomy by splitting the stack into Data and Code Stacks [7][12], known as dual stack technique. As the name implies, the Data Stack will be storing the data content and the Code Stack will be storing the Return Address, and in some implementations the Stack Frame Pointer (SFP) will also be saved [12]. Since there are two stacks that need to be handled, a new Code Stack Pointer (CSP) will be introduced to manage the movement of the CS, whist the current Stack Pointer will be pointing to the original Stack.

The solution using the dual stack requires compiler modification and in some implementation requires new design of hardware to host the new stack. Kugler [12] has chosen to use the current stack to hold the CS, but Jun Xu used the current stack as DS and proposed a new stack to manage the new CS. This technique requires a new mechanism of saving and retrieving information from the new stack which involve changes in some instruction semantics especially in the calling and returning instructions.

The main challenges of this technique is the backward compatibility with existing application that contain inline assembly [12] embed in the code. The growth of the stack should be carefully calculated if both the stack resided in the same area. Assembly language programmer need to pay attention to the movement of the information onto the second stack especially in using the push and pop instructions that may damage the new stack area.

The approach that we are choosing is similar as in the dual stack implementation, except that the DS will be stored in the Alternative Stack that uses the Windows File Mapping to host the data. The Alternative Stack is a piece of software that is loaded in the system and provides the DS service to the caller program through specific event. This approach offers backward compatibility with the older window version and it is easy to be implemented. The biggest challenge of this approach is during the movement of large amount of string data into the Alternative which is costly the application performance.

3. The Alternative Stack Architecture

The current dual-stack approach only stores the stack information and exposes the stack data which is vulnerable to attack. The alternative stack is a piece of software that stores the stack data into a secured storage. The Alternative Stack software is divided into 3 main components:

1. Communication Module
2. Data Module
3. Loader Module

The Communication Module is responsible for delivering the data to and from the Alternative Stack whenever the event is triggered by the calling program. Once the event is triggered, the Communication Module will fetch the data packet which consists of the Field Identification or FieldID, command code and its data content and are passed to the Data Module. Based on the command, the Data Module will perform saving or writing to the Alternative Stack. During the saving operation the data will be XOR'ed with a random number assigned during the loading of Alternative Stack. The Alternative Stack comprise of a link-list of pre-allocation variable field area that is created by the Loader Module during the loading of the Alternative Stack program. The Alternative Stack linked-list variable slot is created using windows file mapping option. Loading of the Alternative Stack depends on which architecture that is running.

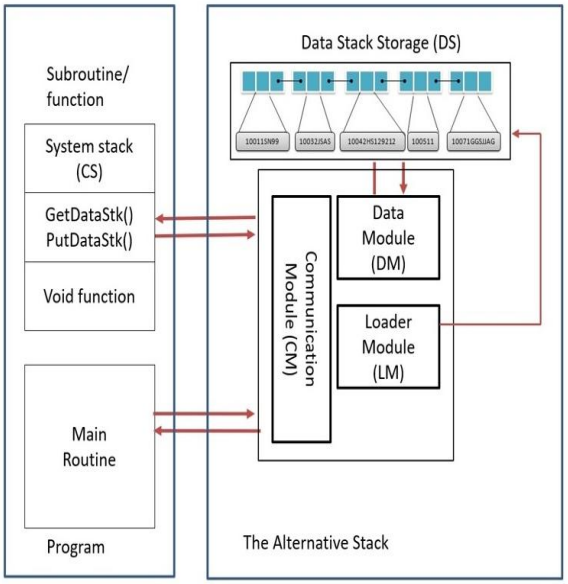


Figure 2: Alternative Stack (Architecture 1 & 2)

For the process type architecture, the Alternative Stack is loaded as a separate process, and in the multi-threaded architecture, it is loaded by the caller application as a thread routine as depicted in Figure 2. We have named the Architecture as ASA1 and ASA2. In the final architecture as illustrated in Figure 3, the Alternative Stack (ASA3) is loaded as a normal subroutine in the caller application.

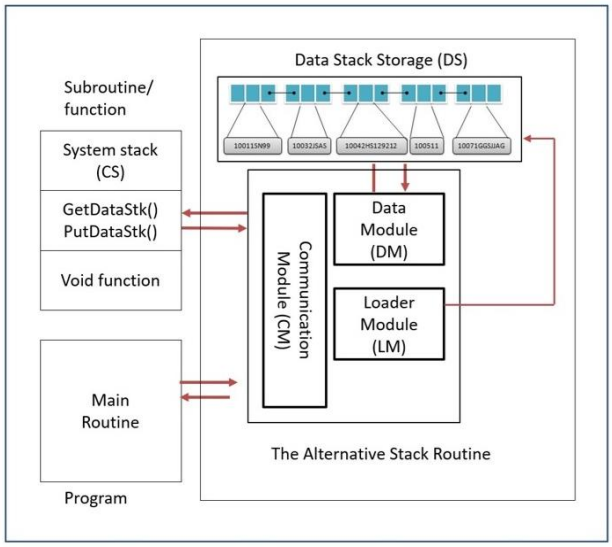


Figure 3: Alternative Stack (Architecture 3)

For the ASA1 and ASA2 the communication module waits and listens to the event created by the caller program. During the retrieval of the data content, upon requesting for the data content, the caller program also will be waiting for a data arrival event. In the ASA3, since Alternative Stack will be called as part of the program routine, the event synchronization is not needed.

4. Testing the Alternative Stack

The main objective of this solution is to ensure that during any attempt of buffer overflow attack, the victimized program should remain "available" and running. The Alternative Stack will be compared with the program compiled using Microsoft Visual Studio for Developer (MVSD) with the security cookie feature turned on. The code that is stuffed with security cookie will be used as the benchmark throughout the test for all the Alternative Stack architecture. It will be a plus point for this Alternative Stack to perform at par with the program containing security cookie and maintain all the three security parameters i.e. CIA will be observed.

The type of testing entails the following:

1. String function
2. Mix functions
3. Buffer overflow attack (during executing string function)

In the section (4.1, 4.2 and 4.3) we will be discussing on the summary of the 3 test scenarios in relation to the Alternative Stack Architecture 3 (ASA3) and in section e (discussion) we will conclude the test result of all the 3 architectures. The ASA3 will be benchmark with a normal program without any security features embedded into it and a program with buffer tagging protecting features switch on. As for the buffer tagging protection, we have selected the Microsoft Security Cookie since we are using the Microsoft Visual C++ Compiler to build all the related programs.

4.1 String Function Testing (Test 1)

The string function test on the ASA3, observed the security parameters and the performance of Alternative Stack in a normal string movement to and from the Alternative Stack module. Observation is made starting from 20 read and write of 20 bytes string until 500 read and write operations. The program with security cookie transfer on an average of 0.000018 seconds per single read and write string and linear regression line of y = 0.0003x + 0.0003 as illustrated in Figure 3. Starting from 0.001 seconds and ending at 0.009 seconds, the average read and write for the Security Cookie in this test is 0.000018 seconds per single read and write.

The Alternative Stack as depicted in Figure 4 started at 0.000 seconds (from data range of 20 to 60 write and read operations) and linear regression line of y = 0.002x + 0.0007 and ends up at 0.005 seconds. The average response time is 0.00001 second per single read and write string operations.

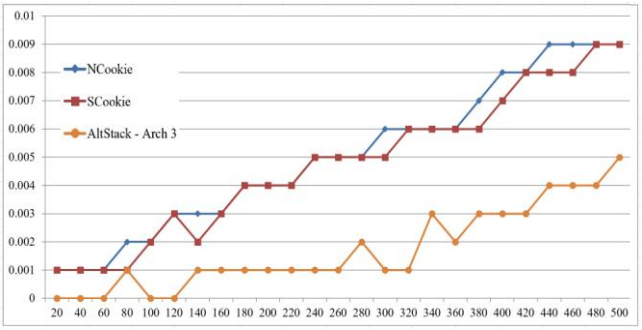


Figure 4: Test 1 (String test) – Normal, Security Cookie and Alternative Stack

4.2 Mix Functions (Test 2)

The purpose of this testing is to observe the program behaviour of ASA3 and Security Cookie in a mix functions environment. The mix functions testing is to ensure that the solution should leverage on the absence of string data which will avoid the addition of extra code in the prologue and epilogue of a specific function. The ASA3 is expected to reflect performance at par with the Security Cookie as well as the performance of Alternative Stack Architecture 1 and 2. In the mix functions testing, application with security cookie started at 0.153 seconds for the first 20 read and write operations with the linear regression line of y = 0.1565x - 0.0008. At 500 read and write operations, the Security Cookie clocked at 3.909 seconds as depicted in Figure 5

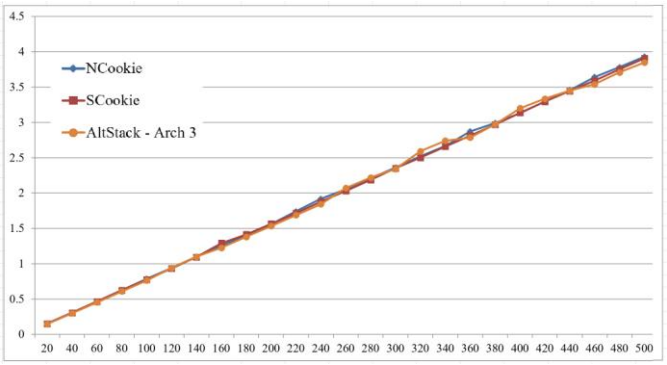


Figure 5: Test 2 (Mixed Functions)

As for the Alternative Stack Architecture 4, it started at 0.146 seconds with linear regression line of y = 0.157x - 0.0087 as illustrated in Figure 5. At the end of 500 read and write operations it clocked at 3.848 seconds.

4.3 Buffer Overflow Attack (Test 3)

The purpose of this testing is to observe the program behaviour during buffer overflow that is triggered whilst transferring of data string to and from the stack. This 3rd test will illustrate the ability of each program to maintain the 3 parameters of the information security, i.e. Confidentiality, Integrity and Availability. This test should be able to differentiate the difference between combating buffer overflow in a live situation and perform process termination, whilst avoiding buffer overflow from happening.

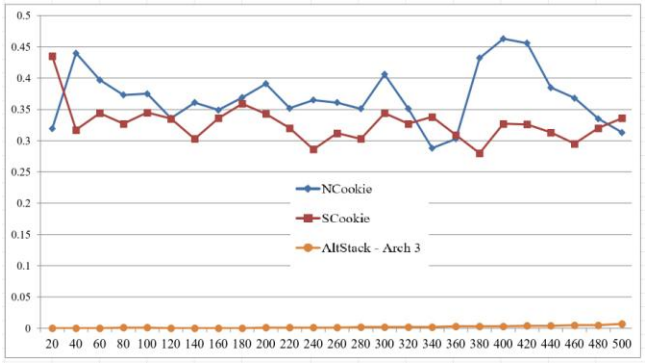


Figure 6: Test 3 (Buffer Overflow)

In this test, when buffer overflow is triggered, the application with Security Cookie terminated the application at average timing of 0.3273 seconds, which is higher than end timing of the same application without buffer overflow at 0.009 seconds. For the ASA3, during buffer overflow, the application started at 0.004 seconds for 20 read and write operations with linear regression line y = 0.0002x + 0.0012 as illustrated in Figure 5. The execution ended at 0.007 seconds for 500 read and write operations which is still lower than the Security Cookies that are terminating the application during buffer overflow.

5. Results and Discussion

In test 1 scenario, Security Cookie started at 0.001 seconds (20 read and write) and managed to complete the test at 0.009 seconds for 500 read and write of data string. On average it took 0.000018 seconds for a single Read and Write of stack data. In this test, ASA3 managed to clock a better performance as compared with the benchmark as illustrated in Figure 7. With the absence of Inter Process Communication, the Alternative Stack processor managed to challenge the performance of Security Cookie. By the end of the 500 read and write strings, the ASA3 completed faster than Security Cookie and a normal program with Security Cookie by 0.004 seconds

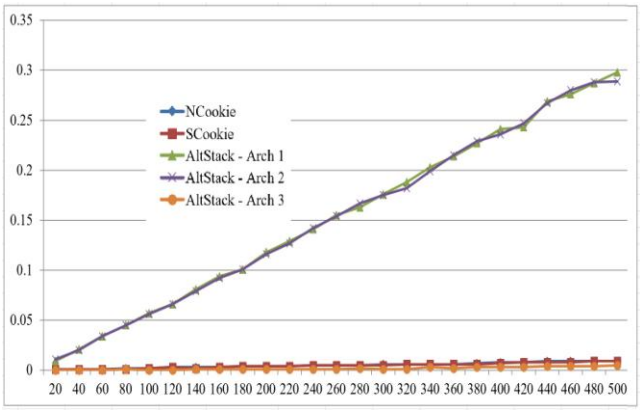


Figure 7: Test 1 (String function) - Response time trends

The performance of ASA1 and ASA2 is about the same. As for the mix functions scenario which is reflected in test 2, the Security Cookie started at 0.153 seconds which is slower than the ASA3 by 0.007 seconds (0.146 seconds). The ASA3 finishes at 3.848 seconds which is slightly faster than Security Cookie by 0.061 seconds (3.909 seconds). This result can be depicted in Figure 8 below. Overall result for the mix functions test shows that all applications perform at almost the same speed. This is also proof that all the Alternative Stack Architectures have no impact on performance in a mix function environment. The ASA3 recorded the fastest performance against all the applications.

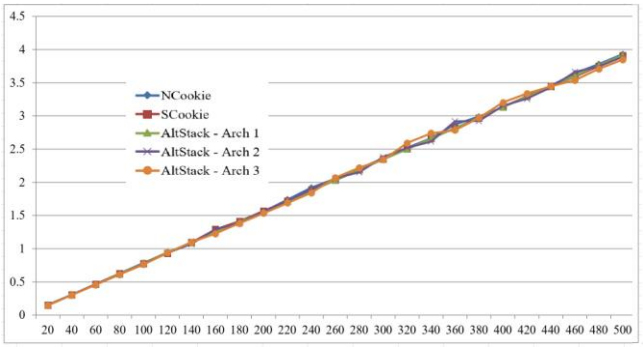


Figure 8: Test 2 (Mix functions) - Response time trends

It is observable that storing of data in the pre-allocated Windows file mapping contributes to the better performance of the ASA3 as compared with storing data in a normal program stack. Both the test 1 and test 2 results are better on the performance against Security Cookie and the other 2 Alternative Stack Architectures. As for the buffer overflow attack in test 3, the Security Cookie terminated the application at an average speed of 0.372 seconds. The ASA3 ended the application at 0.007 seconds, followed by Architecture 1 at 0.185 seconds and Architecture 2 at 0.2 seconds. The result can be depicted in Figure 10 below. During buffer overflow attack, the performance of Architecture 3 still recorded as the fastest solution among the other three Architectures.

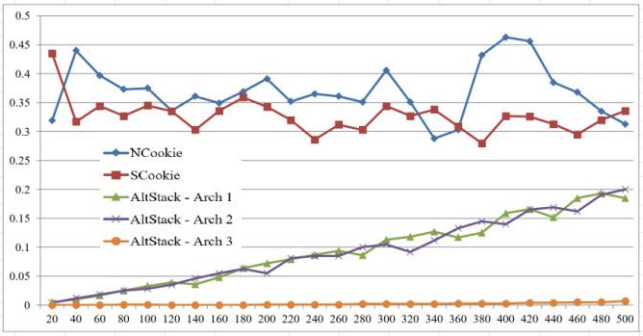


Figure 9: Test 3 (Buffer overflow) - Response time trends

The test results prove that all the Alternative Stack Architectures comply with the three (3) security parameters, especially during buffer overflow attack on the buffer stack as illustrated in the Security Parameter Table shown in Table 1. The significant difference between the four architectures are in the performance. In this context, the ASA3 leads the chart as shown in test 1, followed by the Security Cookie and the other 3 Alternative Stack Architectures during the normal application execution. The ASA3 again leads the mix functions test results by a few seconds, but the timing is not significant as though all the applications perform at the same speed as depicted in Table 2.

Table 1: Security Parameter Table

Stack Solution	Test 1 String function	Test 2 Mix functions	Test 3 Buffer Overflow
Normal application with no Security cookies	A	A	
Application with Security Cookies	IA	IA	I
Alternative Stack – Architecture 1	CIA	CIA	CIA
Alternative Stack – Architecture 2	CIA	CIA	CIA
Alternative Stack – Architecture 3	CIA	CIA	CIA

C – Confidentiality
I – Integrity
A – Availability

Table 2: Performance Test Summary

Test Name	Normal program with no Cookie (Seconds)	*Program compiled with Security Cookie (Seconds)	Alternate Stack – Architecture 1 (Seconds)	Alternate Stack - Architecture 2 (Seconds)	Alternate Stack – Architecture 3 (Seconds)
Test 1 (String function)	0.009	0.009	0.298	0.289	0.005
Test 2 (Mix functions)	3.932	3.909	3.914	3.89	3.848
Test 3 (Buffer Overflow)	0.313	0.336	0.185	0.2	0.007

*Benchmark – Security Cookie

The ASA3 performs at a constant speed during all the tests indicating that this architecture can be used as an alternative solution to avoid buffer overflow from attacking the return address of a program. The change of architecture from loading the Alternative Stack processor in multi process (Architecture 1) or multi thread solution (Architecture 2) to single thread solution created a drastic impact on the performance of the application. The absence of process synchronization boosted up the Alternative Stack performance.

In the application that is compiled with security cookie, the transfer of string was done via a pointer to the array of string. The Alternative Stack used windows memory map to store the link list of pre-allocated Alternative Stack storage, and it was observed that this contributed to improve the reading and writing of string performance. Storing the Data Stack into another memory segment in Windows File Mapping reduced the possibility of data corruption.

6. Conclusion

With the use of Alternative Stack, passing of data through function parameter is no longer a requirement, since data that is needed by a specific function can be stored and retrieved using the Alternative Stack. The function can now be defined as void function, hence it could reduce the possibility of the programmers from making unintentional mistakes. This type of function is suited for Dynamic Link Library (DLL) implementation which can be applied for the late binding mechanism in creating the loosely coupled reusable routine.

In conclusion, the ASA3 offers a new alternative method of storing data stack information as it could match the performance of programs that are compiled with security cookie whilst avoiding the buffer overflow attack on the stack and henceforth could meet the overall objective of this research.

References

[1] Aurélien Francillon, Daniele Perito and Claude Castelluccia. Defending embedded systems against control flow attacks. In Proceedings of the first ACM workshop on Secure execution of untrusted code, 2009 (SecuCode '09) p 19-26.

[2] Crispin Cowan, Perry Wagle, Calton Pu,Steve Beattie, and Jonathan Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. DARPA Information Survivability Conference and Exposition (DISCEX '00)., 2000. Vol. 2. p.119 - 129.

[3] Eugen Leontie, Gedare Bloom, Olga Gelbart, Bhagirath Narahari and Rahul Simha. A Compiler-Hardware Technique for Protecting Against Buffer Overflow Attacks, 2009. URL:https://www.seas.gwu.edu/~simha/research/HWStack.pdf, 07-12-2016.

[4] Gerardo Richarte. Four different tricks to bypass StackShield and StackGuard protection, 2002. URL: https://www.cs.purdue.edu/homes/xyzhang/fall07/Papers/ defeat-stackguard.pdf, 26-09-2017.

[5] Hiroaki Etoh. GCC extension for protecting applications from stacksmashing attacks. URL : https://www.researchgate.net/publicaton/243483996_Gcc_extension_for_protecting_applications_from_stack-smashing_attacks

[6] INTEL. Control-Flow Enforcement Technology Preview, 2016, Jun 2017. Rev 2.0. URL : https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf, 22-08-2017.

[7] Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel and Ravishankar K. Iyer. Architecture Support for Defending Against Buffer Overflow Attacks, 2002. URL : https://www.ideals.illinois.edu/bitstream/handle/2142/74493/B53-CRHC_02_05.pdf?sequence=2,22-08-2016.

[8] K. Piromsopa and R.J. Enbody. Secure Bit: Transparent, Hardware Buffer-Overflow Protection. In IEEE Transactions on Dependable and Secure Computing, Vol 3(4), 2006. pp.365-376.

[9] Marc L. Corliss, E. Christopher Lewis and Amir Roth . Using DISE to protect return addresses from attack. IN: ACM SIGARCH Computer Architecture News - Special issue: Workshop on architectural support for security and anti-virus (WASSA) Homepage archive, Vol 33(1), March, 2005.pp 65 – 72.

[10] Saravanan Sinnadurai, Qin Zhao and Weng-Fai Wong. Transparent Runtime Shadow Stack: Protection against malicious return address modifications, 2008;

[11] Thurston H.Y. Dang, Petros Maniatis and David Wagner. The Performance Cost of Shadow Stacks and Stack Canaries. IN : ASIA CCS '15 Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, 2015. p- 555-566 .

[12] Tilo Muller and Christopher Kugler. SCADS: Separated Control- and Data-Stack. IN: 10th International Conference on Security and Privacy in Communication Networks September 24-26, 2014.

[13] Tzi-cker Chiueh and Fu-Hau Hsu. RAD: A Compile-Time Solution to Buffer Overflow

[14] Attacks. In Proceedings of the 21st International Conference on Distributed Computing

[15] Systems (ICDCS '01), Mesa, AZ, April 2001. SUNY Stony Brook.

[16] Vendicator, "StackShield: A stack smashing technique protection tool for Linux," Jan. 08, 2000.

[17] Yongdong Wu. Enhancing Security Check in Visual Studio C/C++ Compiler. In WRI World Congress on Software Engineering, 2009, Volume: 4. (IEEE publication), p 109-113.