RUNTIME DEFENSE AGAINST CODE INJECTION ATTACKS USING REPLICATED EXECUTION

A THESIS
Submitted by

JISHNU T B110233CS
SHAHAS K P B110193CS

In partial fulfilment for the award of the degree of

BACHELOR OF TECHNOLOGY
IN
COMPUTER SCIENCE AND ENGINEERING

Under the guidance of
POURNAMI P N

DEPARTMENT OF COMPUTER ENGINEERING
NATIONAL INSTITUTE OF TECHNOLOGY CALICUT
NIT CAMPUS PO, CALICUT
KERALA, INDIA 673601

May 15, 2015

ACKNOWLEDGEMENTS

We wish to express our gratitude to Mrs. Pournami P N, Department of Computer Science and Engineering , National Institute of Technology, Calicut for guiding us with our topic and for her criticism and suggestions. We also wish to thank Muralikrishnan K and Vinod Pathari, Associate Professors from CSE Department for their technical guidance.

We also take this opportunity to thank Abdul Nazeer, Head of Department, CSE for providing us with the venues to improve upon our work.

We would also extend our gratitude to all the Faculty members of Computer Science and Engineering Department, National Institute of Technology, Calicut, for their thoughts and suggestions on our project.

JISHNU T
SHAHAS K P

DECLARATION

"I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgment has been made in the text".

Place:	**Signature :**
Date:	**Name :**
	Reg.No:

CERTIFICATE

*This is to certify that the thesis entitled: "***RUNTIME DEFENSE AGAINST CODE INJECTION ATTACKS USING REPLICATED EXECUTION***" submitted by Sri/Smt/Ms*

JISHNU T B110233CS
SHAHAS K P B110193CS

to National Institute of Technology Calicut towards partial fulfillment of the requirements for the award of Degree of Bachelor of Technology in Computer Science Engineering is a bonafide record of the work carried out by him/her under my/our supervision and guidance.

Signed by Thesis Supervisor(s) with name(s) and date

Place:
Date:

Signature of Head of the Department

Office Seal

Contents

Abstract

This project is aimed at implementing an intrusion detection system(IDS). An intrusion detection system (IDS) is an application that monitors a process in its execution and raises an alarm in case of any undesired or malicious activity trying to take over the process. The mechanism put forward in our project is referred to as Multi-variant execution. In this mechanism, different versions of the same process are generated from the source code at the compilation step. Slight differences among them may be noted during their execution. These versions are called *variants*. Under normal execution conditions, the variants are expected to have identical behaviour and if the process is under any attack, there are notable differences in the execution behaviour of variants. There is a monitoring process (multi-variant monitor), which serves this purpose. This monitor compares the behaviour of variants at certain points of execution, referred to as *synchronization points*. In this project, such comparison takes place at the level of system calls and behaviour refers to certain parameters associated with a system call, the divergence of which causes an alarm to be raised. Also, this monitor runs completely in user space.

Figures

Chapter 1

Problem Definition

Our objective is to generate multiple versions (*variants*) of a process during compilation, and then to execute these versions using a multi-variant monitor. The variants differ in certain aspects during execution. The multi-variant monitor compares the behaviour of variants at certain points of execution, called synchronization points. The monitor, running completely in user space, has to execute all the variants and generate an output as if the process is executing independently on its own. Whenever a divergence among the behaviour of variants is detected at synchronization points, an alarm is to be raised, which serves as a method for intrusion detection.

Chapter 2

Introduction

Vulnerabilities have always been a major concern in the software field. A large number of software packages are still written using C and C++, instead of safe programming languages. Writing safe and secure programs using these languages is generally difficult but they cannot be avoided due to their provisions for high performance. Automated removal of software vulnerabilities has thus become a major challenge. Many techniques have been developed to eliminate such vulnerabilities, but an ultimate solution has not yet been devised. Modern static-analysis tools are unable to detect all errors due to lack of run-time information. On the other hand, the efficiency of dynamic tools are also limited, because they do not have a reference for comparison so as to detect any misbehaviour. Also, the large amount of new code written every year has caused an increase in the number of vulnerabilities. Multi-variant code execution, proposed in this project, is a technique that prevents execution of any malicious code and considers many of the problems mentioned above.

Chapter 3

Literature Survey

Our basic objective is to develop an Intrusion Detection System(IDS) in which multiple variants of a single process are executed by the monitor, which compares the variants at the level of system calls. Any deviation among their behaviour will lead to an alarm. As a first step towards the implementation of IDS, we had to conduct a study on the debugging facilities available in linux. [1] and [6] have given us an idea on such available facilities and we have used *ptrace* for the purpose.

Then we concentrated on various types of attacks possible on a process. We got details on buffer overflow attacks from [2], which helped us to be familiar with the steps involved in its demonstration. This study also gave us an idea about the type of variants to be chosen in our multi-variant execution. [3] and [4] gave an overall picture on the the development of a multi-variant system and the possible complications which may arise in the process, while [5] explores randomization possibilities. In our implementation, we have used only two variants, which have different virtual address spaces during execution.

Chapter 4

Design Stages

- **[I]Variant Generation**

  Two or more slightly different versions of the same program, called variants are generated. Such small differences among the variants can be observed during execution. In spite of these differences, the variants remain identical during execution as far as their behaviour is concerned. The divergence in their behaviour occurs only when there is an attack.

- **[II]Multi-variant Monitoring Process**

  It is a user process which executes the variants, generated as output of stage I.

  Input : Path of the directory containing executable variants of a process

  Output : Output of the original source code if there is no divergence among the behaviour of variants and an alarm is raised if there is any divergence.

Chapter 5

Work Done

5.1 Overview

(1) The multi-variant monitoring process, mentioned in stage II has been implemented.

(2) A sample set of source codes is constructed. For each source code from the set, two variants are generated from the code at the compilation stage, which are successfully executed by the multi-variant monitor. The monitor is able to detect stack overflow attack during the testing phase.

5.2 Implementation of Multi-Variant Monitor

We have already gone through an outline of the two main stages in the Design Stages section. The number assigned to the stages has the objective of highlighting the two major and broad phases in our implementation and does not necessarily indicate the order in which they are implemented. In our approach, we have started with stage II and moved to stage I only after stage II is implemented to a certain extent.

Stage II is basically associated with the design and implementation of the multi-variant monitor.

5.2.1 The Multi-Variant Monitor

The duty of our multi-variant monitor is to control the states of variants and to make sure that the variants are complying to a defined set of rules. In our implementation, these rules are formulated as defined in the work by Babak Salamat on multi-variant execution[6]. To understand the functionalities of monitor, we have to be familiar with the following terms in our context:

- *synchronization*: It means that our independently executing variants are to join up or handshake at a certain point and give control to the monitor to carry out a certain sequence of action. Such a point is said to be a *synchronization point*
- *granularity*: It is an indicator of the extent to which we are making the comparison of the behaviour among variants.

We use a monitoring technique that *synchronizes* program instances at the *granularity* of system calls. This is sufficient as any kind of outside effects on a process are prevented in modern operating systems, unless a system call is invoked by the process.

Our multi-variant monitor runs completely in user-space. The monitor is a process invoked by a user and receives the paths of the executable files that are to be run as variants. The monitor creates one child process per variant and starts executing them. Variants are allowed to run without any interruption as long as they do not require any resource from outside their process spaces. Whenever a system call is issued by a variant, the monitor takes over the control and the variant is suspended. The monitor then attempts to *synchronize* the system call with the other variants. The monitor determines if the variants are in complying state based on the following rules[6]. "If p_1 to p_n are the variants of the same program p, they are considered to be in conforming states if the following conditions hold at every *synchronization point*:

- All the variants have invoked the same system call.
  $\forall s_i, s_j \epsilon S : s_i = s_j$ where $S = \{s_1, s_2, ..., s_n\}$ is the set of all invoked system calls at the synchronization point and s_i is the system call invoked by variant p_i .

- There is an *equivalence* among the arguments of system calls invoked by variants.
  $\forall a_{ij}, a_{ik} \epsilon A : a_{ij} \equiv a_{ik}$ where A $= \{a_{11}, a_{12}, ..., a_{mn}\}$ is the set of all the system call arguments encountered at the synchronization point, a_{ij} is the i^{th} argument of the system call invoked by p_j and m is the number of arguments passed to the system call encountered. The argument equivalence operator is defined as:

  a $\equiv b \Leftrightarrow \{$ $iftype \neq buffer$: $a = b$
  $else$: $content(a) = content(b)$

  with type being the type expected for this argument of the system call and *content* refers to the buffer content."

5.2.2 System Call Execution

Our system of monitor and variants executed must act as if any one of the variants is running on the host operating system in a conventional manner. It is our monitor, which provides this feature by running certain system calls on behalf of the variants and providing the variants with the results. We have to study the system calls of the host operating system (Linux in our case) one by one and consider the types and the number of possible arguments that can be passed to them. Based on the effects of these system calls and their results, we have to specify which ones are to be executed by the variants and which ones have to be run by the monitor. As of now, we have considered the following categories of system calls and implemented a total of about 50 system calls, assuring that there is at least one system call from each category :

- When system calls like *write*, *mkdir*, *rmdir*, *chmod*, *create* are encountered, the monitor executes the call on behalf of the variants and passes the return value to variants. The variants skip this call.
- In case of calls like *chdir*, the monitor as well as variants have to execute them.
- Calls like *getppid* has to be executed only by the variants and need not be executed by the monitor.
- For system calls like *getpid*, the call has to be executed by one of the variants and the monitor must pass the result to each of the remaining variants.

5.2.3 Skipping System Calls

This functionality of our monitor is also implemented using methods mentioned in the work by Babak Salamat[6]. Debugging facilities of Linux (*ptrace*) are used to implement the monitor. Depending on the first argument of *ptrace*, a traced process is paused and the tracer is notified. We pass `PTRACE_SYSCALL` to *ptrace* which causes the variants to be paused when a system call is invoked by them or when they receive a signal. The monitor is notified twice per system call, after which monitor can manipulate their contexts :

(1) *At the beginning of the call* : The monitor checks the system call issued and its arguments at the first notification. After ensuring that the variants are complying with our defined rules, the system call is executed. In Linux *ptrace* implementation. we have to perform a system call once a program variant ha initiated a system call. However, if the system call is to be executed only by the monitor, the variants must skip this call. In `x86_64` Linux the system call number is passed in RAX, while in x86, it is passed in EAX. When the monitor needs to make the variants skip the system call, the monitor changes the value of RAX/EAX on receiving the first notifications. In our implementation, the monitor puts system call number corresponding to *getppid* in RAX to make the variant run *getppid* instead of the system call requested initially. After the system call number is replaced, variants are resumed and the operating system runs the new system call replaced by the monitor.

(2) *When the kernel has finished executing the system call handler and has prepared return values* : When system call execution by the variants is finished, the monitor receives the second notification and can replace the results of the system call returned to the variants when needed. Most of the system calls return results in RAX(in `x86_64`). All the system calls considered till now falls in this category.

.

5.2.4 Monitor-Variant Communication

The monitor executes the variants as its own children and traces them. The register values of the children are obtained using *ptrace* with `GET_REGS` as argument while `SET_REGS` can be used to set values of these registers. Since the monitor is executed in user mode, it is unable to read data from or write data to the memory spaces of variants directly. For the comparison of contents in the indirect arguments passed to the system calls, the monitor may need to read content from the memory of the variants. Also, the monitor has to write to their address spaces, if a call executed by the monitor and skipped by the variants returns results in memory. We can read from the memory of the processes by calling *ptrace* with `PTRACE_PEEKDATA` as argument when the variants are suspended[6]. We have used this method in our implementation, on account of its simplicity though it is less efficient compared to some other methods.

5.2.5 White List for *exec*

White list is a text file which contains the names of executable files with complete path, which have the permission to be executed by user variant programs. Monitor checks arguments of *exec* system call invoked by variants.The argument list contains executable file name with complete path, which must belong to the white list passed to the monitor. If it does not belong to the white list, *exec* system call is denied and monitor will shut down the process with warning.

5.3 Variant Generation and Multi-Variant Execution

A set consisting of a wide range of source codes is constructed. These are the source codes whose variants are to be executed by our multi-variant monitor. For each of the source code, two variants are generated at the compilation step.

5.3.1 Variant Execution and Address Space Randomization

We execute these variants by enabling address space randomization. It is the duty of the multi-variant monitor to execute these variants. During execution, these variants differ in the positions of key data areas of the program, including their base and the positions of the stack, heap and libraries in their address spaces. Under normal circumstances, the monitor gives the impression that only one process is running and produces the corresponding output.

5.3.2 Stack Overflow and its Detection

We just had an outline of the framework in which a process will be running in our monitoring mechanism. Now we will look into a sample stack overflow attack. Suppose an attacker attempts to launch a stack overflow attack, by corrupting the return address of a function and setting the same to a value of his choice. If the program is running on its own as a single process, the attacker can corrupt the return address through his input, once he gets the knowledge about the process' address space. In our case, the monitor acts as the mediator and the attacker has to deal with two variants. The malicious input from the attacker is accepted by the monitor, which then passes the value to variants. Even if the attacker gets details about the address spaces of the variants, he can launch an attack only on one of the variants. Suppose he successfully corrupts the return address in one of the variants. In the other variant, the same address points to a different location, by virtue of address space randomization. Consequently, the variants are no longer synchronized and there is a high possibility for a system call mismatch, by name or arguments, between the variants. This will be detected by our monitor.

Chapter 6

Statistical Overhead Analysis

6.1 Experimental Results

An analysis on the overhead generated as a result of multi-variant execution has been carried out. CPU times for write system calls are taken into account in our analysis. For this purpose, a set of sample programs is constructed, each making a different number of system calls. Each program in the set is executed in two environments - first outside the monitor and then within our multi-variant monitor, after generating two variants of each program. Our analysis consists of a comparison on the overall CPU times(average of multiple instances) taken for system calls in the two cases. Based on the data obtained from this analysis, two graphs are plotted :

(1) Average CPU times encountered in normal environment against number of system calls

(2) Average CPU times encountered in monitored environment against number of system calls

In either case, initial overheads are neglected as we are interested only in the overhead associated with system calls.

The results obtained from the graphical analysis are given below:

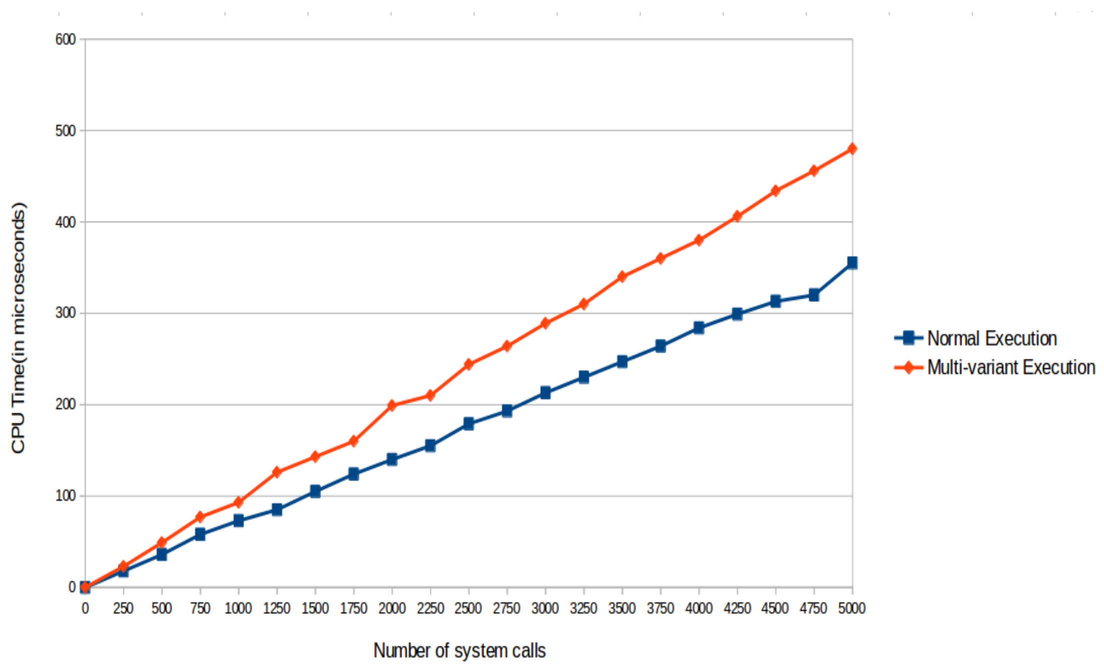

Figure 6.1: Graph showing data from analysis

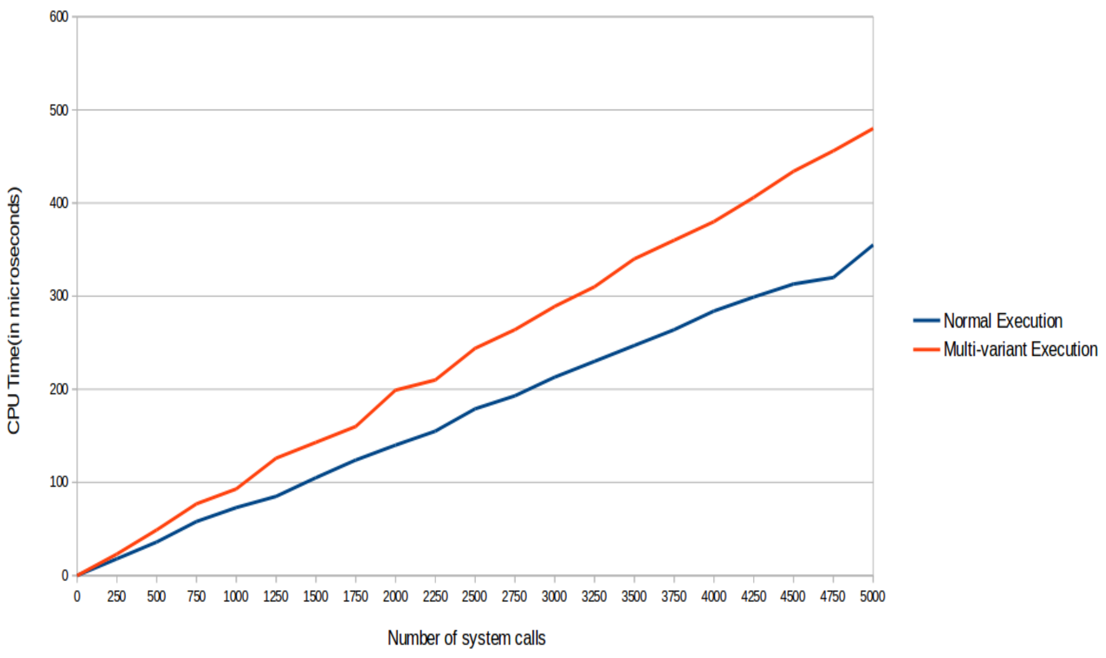

Figure 6.2: Graph without highlighting data

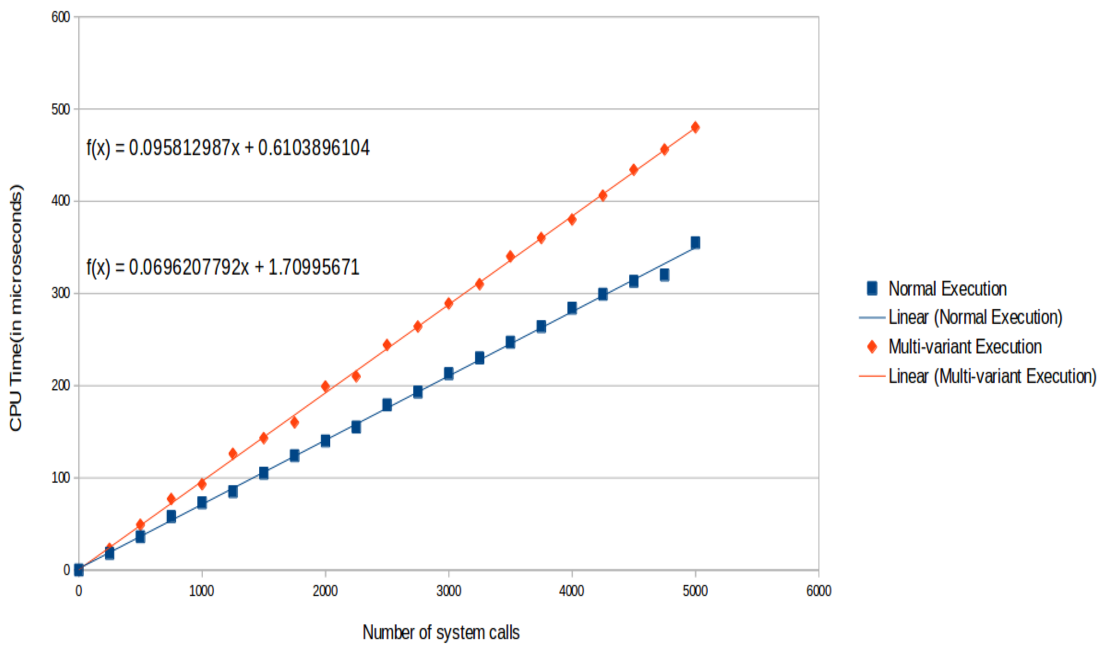

Figure 6.3: Approximated straight line graph

From the straight line approximations of the original plots, we try to derive a mathematical conclusion on the overhead in multi-variant execution.

Equation of straight line 1(normal environment) : 0.070x + 1.710 ......(i)

Equation of straight line 2(within monitor) : 0.096x + 0.610 ......(ii)

CPU time per system call in normal execution(in microseconds) = Slope of line (i) = 0.070

CPU time per system call in multi-variant execution(in microseconds) = Slope of line (ii) = 0.096

Additional overhead per system call in multi-variant execution = (0.096 - 0.070) microseconds = 0.026 microseconds = (0.026/0.070)*100 % = 37 %

6.2 Explanation

The above overhead is mainly contributed by the following factors:

(1) The waiting time encountered by the monitor while it waits for a system call to be invoked by both the variants.

(2) The overhead created by monitor-variant communication. In our experiments, the string to be printed must be communicated from the memory of variants to the monitor memory, which contributes to the overhead.

Chapter 7

Problems and Challenges Faced

7.1 Variant Generation

As of now, we are executing only two variants using our multi-variant monitor. We have tried to generate one more variant with reversed stack growth. However, the process involved modification of gcc compiler. We could not proceed due to the complicated steps in the process.

7.2 Shared Memory In Monitor-Variant Communication

One of the giant leaps towards reducing multi-variant execution overhead would be implementing shared memory communication between monitor and variants. Such an implementation requires establishing a shared memory between monitor and each of the variants as well as attaching them to the shared memory. Since the variants do not contain any code concerning shared memory, we have to modify the variant code from the monitor. This process seemed to be too complicated beyond a certain stage.

7.3 Study on System Calls

There are a number of system calls to be studied in detail before their implementation in the monitor. However, the study has been conducted on a maximum number of system calls.

Chapter 8

Future Work

8.1 Improving Communication Efficiency

In our implementation, *ptrace* facility is used for communication between monitor and variants. Though it is easy to implement, it is less efficient compared to other methods available. Implementing shared memory between monitor and variants can significantly reduce communication overhead.

8.2 Demonstration in Real World Scenarios

A client-server architecture can be used for demonstrating the efficiency of the system, thereby simulating a real world scenario.

8.3 Increasing Number of Variants

Different types of variants, other than the ones used in our implementation, can be generated for the purpose of detecting other possible attacks.

Chapter 9

Conclusion

A version of the monitoring mechanism has been successfully implemented and tested against a set of source codes. As of now, two variants are generated from the source code of a process. The monitor is able to execute all the variants in the directory given as input as if only one process is running. The monitor in its current form is also able to detect stack overflow attacks. For convenience purposes, the attack input is now issued from the system in which the process is executing, though it is obvious that this is not the real scenario. The monitor may be tested against a client-server architecture in future. We have also developed an idea on the overhead generated by our multi-variant execution. Measures can be taken in future to reduce its value, which will result in an improved performance of the system.

Bibliography

[1] K. Knowlton. A combination hardware-software debugging system. IEEE Transactions on Computers, 1968.

[2] Y. Park Z. Zhang and G. Lee. Microarchitectural protection against stack-based buffer overflow attacks. IEEE MICRO, pages 62–71, 2006.

[3] T. Jackson A. Gal B. Salamat and M. Franz. Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space. Proceedings of the European Conference on Computer Systems, 2009.

[4] M. Drinic and D. Kirovski. A hardware-software platform for intrusion prevention. MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture, pages 233–242, 2004.

[5] A. Keromytis G. Kc and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. Proceedings of the ACM Conference on Computer and Communications Security, 19(3):270–280, May 2003.

[6] Babak Salamat. Multi-variant execution: Run-time defense against malicious code injection attacks. 2009.